INTRODUÇÃO À ARQUITETURA E DESIGN DE SOFTWARE

UMA VISÃO SOBRE A PLATAFORMA JAVA

PAULO SILVEIRA
GUILHERME SILVEIRA
SÉRGIO LOPES
GUILHERME MOREIRA
NICO STEPPAT
FABIO KUNG

Prefácio de Jim Webber

"A nossa avó Céres"
– Paulo Silveira e Guilherme Silveira

"A minha esposa, meus pais e meu irmão"
– Sérgio Lopes

"A meus pais e esposa"
– Guilherme Moreira

"A meus amados pais"
– Nico Steppat

"A minha amada esposa e família"
– Fabio Kung

Agradecimentos

Depois de dois anos ministrando o curso de Arquitetura e Design na Caelum, além de cinco anos com um blog de mais de duzentos artigos técnicos, parecia fácil escrever sobre um assunto que faz parte do nosso dia a dia.

Terrível engano.

Em vez dos seis meses previstos, passaram-se dois anos. Dois anos trabalhando de forma a tornar acessível práticas e trade-offs que aparecem com frequência na plataforma. São problemas que os desenvolvedores passam a enfrentar rapidamente ao ganhar mais experiência e familiaridade com Java.

Juntar esse conhecimento não seria possível sem a existência do GUJ.com.br, criado em 2002, no qual os usuários já trocaram mais de um milhão de mensagens. Nós, autores, participamos ativamente do fórum, o qual não atingiria essa magnitude sem a ajuda do Rafael Steil.

Também não seria possível organizar essas ideias sem a colaboração de diversos amigos que as revisaram e discutiram os temas conosco: Lucas Cavalcanti, David Paniz, Cecilia Fernandes, Alberto Souza, Bruno de Oliveira, Fábio Pimentel, Vinicius Roberto e Adalberto Zanata.

Agradecemos aos que participaram e participam da elaboração do curso FJ-91 na Caelum: Adriano de Almeida, Anderson Leite, José Donizetti e Douglas Campos.

Uma especial menção ao Maurício Aniche, que, além de realizar uma minuciosa revisão, nos incentivou a todo momento a melhorar o texto e a cortar o que não fosse essencial para o livro.

Sobre os autores

Paulo Silveira é bacharel e mestre em ciência da computação pela USP e cofundador da Caelum. Possui experiência com desenvolvimento web na Alemanha, passando pelo IBOPE e por dois anos como instrutor da Sun Microsystems. É um dos fundadores do GUJ.com.br, editor técnico da revista MundoJ e criador do VRaptor.

Guilherme Silveira é líder técnico e cofundador da Caelum. Trabalhou com Java e Tibco na Alemanha, comitter do XStream e criador do Restfulie. Concentra seus estudos em serviços REST. Programador experiente, participou das finais mundiais de programação no Japão e no Canadá do ICPC pelo time da USP e é arquiteto Java certificado (SCEA5).

Sérgio Lopes é coordenador na unidade São Paulo da Caelum. Programa desde 2000 e desenvolve em Java desde 2003. É arquiteto Java certificado (SCEA5), moderador do GUJ e participante ativo da comunidade através de palestras, projetos, cursos e artigos.

Guilherme Moreira é instrutor e consultor pela Caelum, onde foca seu trabalho principalmente com Java desde 2005, com projetos internos e também diretamente nos clientes. Formado em Processamento de Dados pela FATEC-SP e arquiteto Java certificado (SCEA 5), escreve artigos para a revista MundoJ, além de ser responsável pela unidade de Brasília da Caelum.

Nico Steppat é engenheiro de computação aplicada pela Fachhochschule Brandenburg. É instrutor, consultor e desenvolvedor há sete anos com Java no Brasil e na Alemanha. Atua na unidade Rio de Janeiro da Caelum e é arquiteto Java certificado (SCEA 5).

Fabio Kung é engenheiro da computação pela Escola Politécnica da USP. Passou por web, Java, Ruby, experiência na Alemanha, instrutor na Caelum, responsável na Locaweb por Cloud Computing e no Heroku em São Francisco. É arquiteto Java certificado (SCEA 5).

Prefácio

Por Jim Webber.

O Arquiteto

Quando penso sobre o termo "arquiteto", é sempre com sentimentos confusos, como suspeito que seja o caso para muitos de nós que trabalham com desenvolvimento de software atualmente. Durante minha carreira, ouvi diversas opiniões diferentes para esse termo, mas, para mim, o que é principal para ser um bom arquiteto posso buscar dos meus primeiros dias como desenvolvedor profissional (ou tentando ser um profissional).

Meu primeiro emprego parecia ser intimidador - após diversos anos de pesquisa acadêmica dentro de um departamento de ciência de computação em uma universidade, entrei em uma empresa que havia comprado recentemente uma startup de middleware Java gerenciada por um grupo de amigos. Esses amigos trabalhavam no ramo de middleware por décadas e "hackeavam" em Java até mesmo antes de ele ter esse nome. Sair da zona de conforto da computação de alto desempenho, com tecnologias as quais eu estava acostumado, e pular para um domínio novo de processamento de transações distribuídas com foco em CORBA e J2EE era realmente intimidador. Apesar de ter um senso razoável de teoria de sistemas distribuídos, eu jamais havia escrito CORBA ou J2EE como usuário, muito menos como desenvolvedor de middleware. Foram tempos complicados.

No laboratório onde comecei a trabalhar, meus colegas ostentavam nomes de cargos baseados em suas capacidades (e, suspeito, longevidade!). Possuíamos engenheiros, engenheiros seniores e arquitetos, todos os quais trabalhavam arduamente em desenvolver um excelente middleware de transação, workflow e mensageria. Como um engenheiro sênior, eu era um pouco menos verde do que os apenas engenheiros, mas ambos recebíamos ordens dos arquitetos - um por time - que eram oráculos. Os arquitetos que lideravam nosso time conheciam o código de trás para a

frente, mas também conheciam seus campos. Era sempre possível confiar que seriam capazes de explicar modelos de transação distribuída complexos e mostrar como diversos padrões e abstrações do código os suportavam. Eles também conheciam as armadilhas e conseguiam detectar cedo escolhas de design e implementação ruins, uma vez que eram capazes de ver à frente, ou pelo menos ver mais longe do que eu conseguia.

Não acho que disse algum dia para esses arquitetos - talvez por causa do meu orgulho juvenil - o quão importante eles foram em delinear meu pensamento e comportamento. Entretanto, sempre aspirei (e algumas vezes sucedi) ser como eles, conhecer a tecnologia profundamente, entender o contexto no qual ela está sendo empregada, guiar e ser guiado pelos colegas de meu time. Essa foi uma aspiração que levei para meu próximo emprego em uma empresa global muito conhecida e admirada no ramo de consultoria de TI.

Ao entrar em minha nova empresa, decidi que já havia visto bastante software e era bom o suficiente em escrever código, e então me declarei um arquiteto. Ao encontrar alguns de meus novos colegas pela primeira vez, fui calorosamente recebido e perguntado sobre qual seria meu papel. Respondi que eu era um arquiteto; algo que pensei que daria um sentido de competência no nível micro da pilha de tecnologias, e no nível macro ao redor da construção de sistemas. Fiquei atônito ao descobrir que fui repreendido por essa resposta e fiquei me perguntando o que acabara de acontecer, afinal, eu não havia entrado para trabalhar com essas pessoas devido a reputação que eles possuíam no desenvolvimento de software e Agile?

Não demorou muito para que eu entendesse o contexto. Meus colegas trabalhavam com consultoria há alguns anos e conheceram muitos "arquitetos experientes". Mas esses não eram o mesmo tipo de pessoa que eu admirava em meu primeiro trabalho; eram arquitetos perigosamente ultrapassados, com pouca habilidade em entregar software, e um talento sem fim para causar confusão e criar políticas corporativas. E eu, aparentemente, tinha me apresentado como um desses. Não era a melhor maneira de causar uma boa impressão aos meus novos colegas.

Ainda assim, durante os 6 anos e nos 10 países em que trabalhei, me descrevi como arquiteto. Eu estava em uma missão de recuperar o valor desse título, e com o passar do tempo obtive algum sucesso. Começamos a pensar em um arquiteto não só como o "melhor programador" ou um "peso morto", mas sim como um papel que considera os detalhes de nível macro em um time que entrega software - como a integração com outros sistemas funcionará, como falhas e recuperações serão trabalhadas, como a vazão será medida e garantida.

Nos melhores projetos em que trabalhei, o papel do arquiteto era complementar ao dos stakeholders. Enquanto o pessoal de negócios prioriza a entrega de funcionalidades, arquitetos priorizam os requisitos não funcionais e ajudam a ordenar a entrega de requerimentos funcionais que minimizam e mantêm características não funcionais. Nos melhores projetos que trabalhei, eu não era o melhor programador da equipe, nem na maior parte das vezes tinha o maior conhecimento das linguagens, ferramentas e frameworks comparado com outros dedicados programadores. Após um tempo engolindo o orgulho, comecei a ver as vantagens inerentes disso.

Como arquiteto, ainda sou alguém que também entende bastante de código, em um nível no qual consigo escrever código com outros programadores (possivelmente melhores do que eu) e ler seu código e testes. Acredito que isso seja fundamental para a arquitetura. Sem a habilidade de se envolver com o código, arquitetos não têm como determinar como sua arquitetura foi adotada por uma solução, nem como eles têm uma maneira direta de pegar feedback de sua arquitetura para melhorá-la. Sem habilidades de software, um arquiteto fica refém dos times de desenvolvimento, o que é um overhead, não um benefício.

Sentado aqui em minha mesa, em Londres, escrevendo esse prefácio, é claro que a noção do conceito de arquiteto continua cinzenta. Até mesmo hoje, arquitetos estão frequentemente no topo da torre de marfim, descrevendo visões grandiosas e impraticáveis através de slides e diagramas UML que não ajudam para tropas nas trincheiras obedecerem como escravos (como se eles fossem!). Mas não precisa ser assim. Aqueles de nós que se preocupam com a melhoria contínua do ramo e se preocupam em escrever código cada vez melhor com os efeitos colaterais positivos que isso pode resultar em nossa sociedade e comunidade têm o dever de tomar o termo "arquiteto" e reapropriar-se dele.

Este livro é um passo nessa direção. Ele trata arquitetos como profissionais do desenvolvimento de software e é fundamental que eles entendam as ferramentas comuns do ramo de desenvolvimento de software. Este livro provê insights nas ferramentas comuns, padrões de desenvolvimento e práticas de design que arquitetos contemporâneos trabalhando em aplicações corporativas possuem. Ele trata de uma gama de disciplinas de desenvolvimento desde código robusto, design de aplicação, dados e integração, que são os principais elementos da maior parte dos sistemas corporativos. Ele não apenas foca em importantes detalhes da linguagem e dos principais frameworks, mas também no nível macro, dando uma percepção mais completa - percepção esta que um arquiteto praticante precisa quando ajuda a guiar um time para o sucesso.

À medida que desenvolvedores procuram se tornar arquitetos, o livro mostra que o foco estreito na última linguagem ou framework da moda é necessário, mas não suficiente. À medida que arquitetos buscam reabilitação, este livro deixa óbvio que não há sucesso sem reaprender as (habilmente demonstradas) ferramentas de construção de software. Enquanto nenhum livro pode torná-lo um arquiteto da noite para o dia, este provê ao menos um modelo do tipo de arquiteto que eu tanto admirava, e espero que sua leitura o inspire a se tornar esse tipo de arquiteto também.

Jim Webber é formado em computação e possui doutorado pela University of Newcastle. Trabalhou por bastante tempo na ThoughtWorks, onde ficou conhecido pelo seu manifesto Guerrilla SOA. Hoje, é Chief Scientist na Neo Technology, empresa por trás do banco de dados orientado a grafos Neo4j

Introdução

Implementação, design ou arquitetura?

Desenhar sistemas é uma tarefa difícil. E, ainda fazer com que sejam escaláveis e performáticos, mantendo uma alta qualidade interna e externa, é um desafio.

No artigo "*A importância de ser fechado*", Craig Larman diz que "*você deve enfrentar suas batalhas de design, sejam elas no nível macroarquitetural ou no humilde campo das instâncias*" [109]. Essas batalhas devem ser contra decisões que atrapalham o crescimento e a futura modificação da sua aplicação, diminuindo o custo de manutenção a longo prazo.

Mas é difícil definir com precisão o que são instâncias ou nível macroarquitetural, as linhas que separam arquitetura, design e implementação são muito tênues. Martin Fowler fala o mesmo no âmbito de patterns logo na segunda página de seu livro"*Patterns of Enterprise Application Architecture*: "*Alguns dos padrões neste livro podem ser chamados arquiteturais, já que representam decisões importantes sobre essas partes; outros são mais sobre design e o ajudam a implementar essa arquitetura. Não faço nenhuma forte* ***tentativa de separar esses dois, uma vez que aquilo que é arquitetural ou não é subjetivo***". [68] Fowler questiona inclusive o uso do termo *arquiteto* como papel. [70] A universidade de Carnegie Mellon possui diversas definições e também aborda a grande dificuldade que é definir a diferença exata entre implementação, design e arquitetura. [51], [1]

Neste livro, **em vez de encontrar respostas e soluções encaixotadas, apresentaremos perguntas e discussões** relacionadas a alguns dos principais tópicos recorrentes em aplicações escritas para a plataforma Java, a fim de alimentar o espírito crítico do desenvolvedor. Ele deve ser capaz de enxergar ambas, as vantagens e as desvantagens; todo o *trade-off* presente em suas decisões.

Dominar uma plataforma, seus principais frameworks e seu funcionamento interno é essencial para que os responsáveis pelo desenvolvimento de uma aplicação conheçam não só as vantagens, mas também as desvantagens de suas decisões. O desenvolvedor que acredita não existir ou desconhece o lado negativo para determi-

nada abordagem sofrerá as consequências de manutenção quando as mesmas surgirem dentro da aplicação.

Com o passar do tempo, a implementação ou o design ou a arquitetura deixará de atender às necessidades dos clientes. Nesse instante, é necessário evoluir esse aspecto defasado, para que volte a ser suficientemente adequado para aquela realidade do sistema.

Como Joel Spolsky afirma, "*pessoas de todo o mundo estão constantemente criando aplicações Web usando .NET, Java e PHP. Nenhuma delas está falhando por causa da escolha da tecnologia*". [191] Mas ele vai além, ao dizer que "*todos esses ambientes são grandes e complexos, e* ***você realmente precisa de, no mínimo, uma pessoa com enorme experiência de desenvolvimento na plataforma escolhida****, porque, caso contrário, você tomará decisões erradas e acabará com um código muito confuso que precisa ser reestruturado*". Joel deixa explícita a necessidade de profundo conhecimento técnico da plataforma escolhida.

Fowler ainda complementa que, diferentemente da engenharia civil, onde há uma divisão clara de habilidades entre quem faz o design e quem faz a construção, com software é diferente. [71] Qualquer desenvolvedor que venha a lidar com decisões de alto nível necessita de um forte conhecimento técnico. Este livro foi elaborado com essa visão: o sucesso da arquitetura de um sistema está fortemente ligado com o design e a implementação.

Levando em consideração essa visão, este livro destina-se àquele que aspira por um papel de arquiteto? Quem seria o arquiteto? O papel do arquiteto é amplamente discutido e, na nossa visão, o arquiteto precisa conhecer profundamente a plataforma, além de ser um excelente e experiente desenvolvedor.

Nas palavras de Martin Fowler, "*o termo arquitetura envolve a noção dos principais elementos do sistema,* **as peças que são difíceis de mudar**. *Uma fundação na qual o resto precisa ser construído*".

[71] Podemos ir além e dizer que essa fundação que constitui a arquitetura, que envolve os principais elementos do sistema, pode sim ser evolutiva e sofrer modificações com o tempo, desde que seu design e implementação permitam, quebrando um pouco a ideia de que o trio arquitetura-design-implementação forme um modelo estritamente piramidal. Este livro abordará muitas questões de design e implementação com o intuito de demonstrar essa possibilidade evolutiva.

A imagem de um arquiteto distante do dia a dia do desenvolvimento da equipe, sem profundo conhecimento técnico e de implementação sobre a ferramenta, que apenas toma as grandes decisões e despacha ordens e diagramas, há muito tempo

está defasada. Mais do que querer ser o isolado arquiteto que apenas despacha ordens, cada vez mais enxergamos que o caminho é ser o líder incentivador da tomada de decisão, além de um exímio desenvolvedor. [190] Parafraseando mais uma vez Martin Fowler, "*(...) o arquiteto deve ser como um guia (...) que é um experiente e capacitado membro da equipe que ensina aos outros se virar melhor, e ainda assim está sempre lá para as partes mais complicadas*". [70] E Larman também não perdoa o arquiteto que não vê código: "*um arquiteto que não está a par da evolução do código do produto, não está conectado com a realidade*", chegando a considerar que o código é o verdadeiro design ou arquitetura do software. [111]

É impossível "mudar" uma arquitetura ou um design diretamente, uma vez que ambos são interpretações do único bem existente: a implementação. Através da mudança na implementação, adquire-se as características arquiteturais ou de design desejadas. **Arquitetura e design são interpretações, visões, leituras críticas de uma implementação** (Figura 1).

Figura 1: Arquitetura e design são interpretações, leituras críticas de uma implementação.

Design é uma forma de interpretar a implementação, analisando e compreendendo as interações entre determinadas partes do sistema, como possíveis impactos que uma mudança em um ponto causa em outro componente que o acessa direta ou indiretamente. Existem diversas interfaces de comunicação entre diversos componentes: o Apache se comunica com o Jetty através do *mod_jk*; uma aplicação Web

se comunica com o Jetty através da Servlet API; já dentro do domínio da aplicação, outras interfaces conectam partes de uma implementação, como uma `List`, um `Calendar`, um `DAO` ou uma `Connection`. Todas elas são analisadas no aspecto de design (Figura 2).
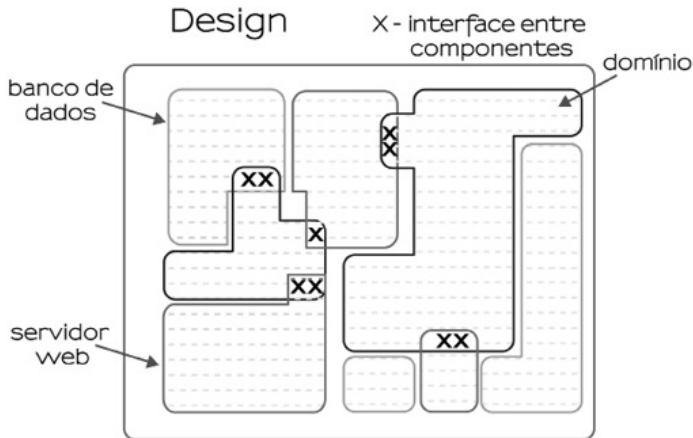
Figura 2: No ponto de vista do design, importam características das interfaces de comunicação entre partes do sistema, seus componentes, em todos os níveis de abstração: desde entre classes até dois softwares distintos.

Implementação não significa somente o código escrito pelos desenvolvedores da equipe atual, mas também o conjunto de escolhas de ferramentas, versões que fazem parte dos ambientes de desenvolvimento, homologação e produção, linguagem utilizada, legibilidade do código, entre outras. Todas essas escolhas modelam a implementação de um sistema, assim como sua qualidade (Figura 3).

Arquitetura é ainda uma outra maneira de ver a implementação, analisando e compreendendo como uma mudança em determinado ponto do sistema afeta o sistema inteiro, e de todas as maneiras possíveis. Por exemplo, caso um ponto qualquer da aplicação passe a ter acesso remoto ao invés de apenas local, isso pode afetar uma outra ponta da aplicação, deixando-a mais lenta. Esse tipo de análise não costuma ser feita quando se pensa no design, já que o design tem uma preocupação mais local. A essa visão mais global do sistema, é dado o nome de arquitetura (Figura 4).
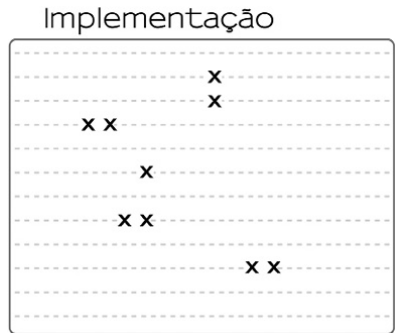
Figura 3: No ponto de vista da implementação, analisa-se o código, as escolhas de versão, linguagem utilizada, legibilidade do código etc.
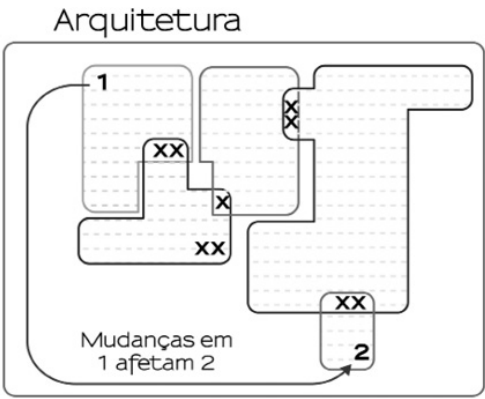
Figura 4: No ponto de vista arquitetural, analisa-se como escolhas de implementação influenciam o sistema inteiro - não como um todo, não como um só, mas por completo.

Com o passar do tempo, como design e arquitetura são diferentes maneiras de interpretar a implementação, é através de uma mudança na implementação que obtém-se as mudanças desejadas na visão de design ou na visão de arquitetura (Figura 5).
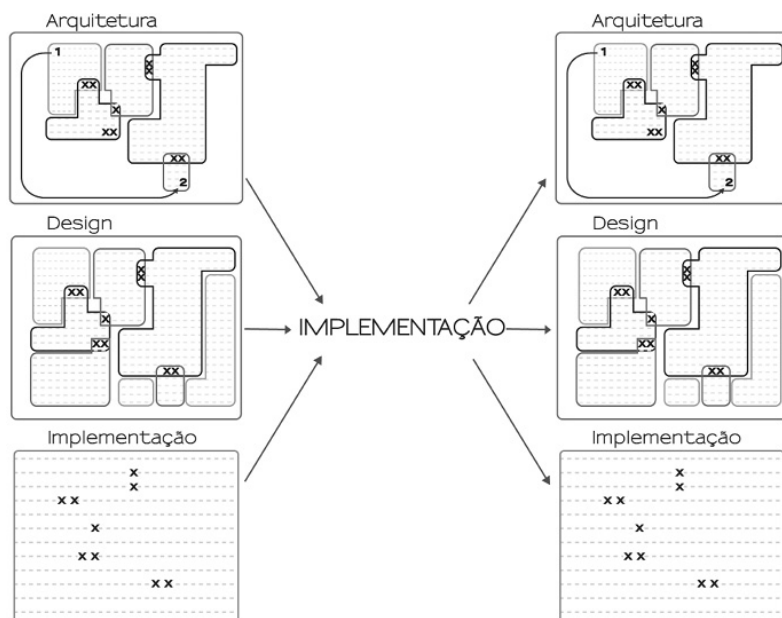
Figura 5: Uma vez que somente implementações são concretas, todo tipo de mudança implica em conhecer a implementação e, alterá-la causa possíveis mudanças nos aspectos arquiteturais e de design.

Uma boa implementação, design ou arquitetura é aquela que permite modificações causando somente um impacto considerado justo a outras partes do sistema, e, ao mesmo tempo, diminuindo as ocorrências de tais situações, minimizando o custo de desenvolvimento e manutenção.

É essa visão de arquitetura e design que tentamos obter neste livro. A plataforma Java é vista com profundidade e suas peculiaridades, desde o campo das instâncias, interfaces e da JVM, até o nível macroarquitetural de layers, tiers e frameworks. A implementação terá papel fundamental aqui, e sua importância é realçada mesmo em discussões arquiteturais de alto nível.

Colhemos nossa experiência ao longo de nove anos de discussões nos fóruns do GUJ.com.br, cinco ministrando o curso de Arquitetura e Design Java na Caelum, além de diversos projetos, consultorias e experiências trocadas com os clientes e parceiros. **Conhecer profundamente as ferramentas é o primeiro passo para poder fazer as perguntas corretas ao enfrentar o cenário de uma nova aplicação**.

Sumário

Capítulo 1

A plataforma Java

Diversas plataformas de desenvolvimento possuem grande penetração no mercado.A plataforma Java atingiu a liderança devido a algumas características relacionadas ao seu processo de evolução e especificação, junto com a participação forte e ativa da comunidade. Conhecer bem o ecossistema Java revela seus pontos fortes e também as desvantagens e limitações que podemos enfrentar ao adotá-la.

1.1 Java como plataforma, além da linguagem

É fundamental conhecer com que objetivos a plataforma Java foi projetada, a fim de entender com profundidade os motivos que a levaram a ser fortemente adotada no lado do servidor. Java é uma plataforma de desenvolvimento criada pela Sun, que teve seu lançamento público em 1995. Ela vinha sendo desenvolvida desde 1991 com o nome *Oak*, liderada por James Gosling. O mercado inicial do projeto Oak compunha-se de dispositivos eletrônicos, como os *set-top box.*[39]

Desde a sua concepção, a ideia de uma plataforma de desenvolvimento e execução sempre esteve presente. O Java idealizado era uma solução que facilitava o

Java é uma completa plataforma de desenvolvimento e execução. Esta plataforma é composta de três pilares: a máquina virtual Java (JVM), um grande conjunto de APIs e a linguagem Java.

O conceito de máquina virtual não é exclusividade do Java. As *Hardware Virtual Machines*, como VMWare, Parallels ou VirtualBox, também muito famosas, são usadas para rodar vários sistemas operacionais ao mesmo tempo, utilizando o recurso de virtualização que abstrai o hardware da máquina. Já a JVM, **Java Virtual Machine**(Máquina Virtual Java), é uma *Application Virtual Machine*, que abstrai não só a camada de hardware como a comunicação com o Sistema Operacional.

Uma aplicação tradicional em C, por exemplo, é escrita em uma linguagem que abstrai as operações de hardware e é compilada para linguagem de máquina. Nesse processo de compilação, é gerado um executável com instruções de máquina específicas para o sistema operacional e o hardware em questão.

Um executável C não é portável; não podemos executá-lo no Windows e no Linux sem recompilação. Mas o problema de portabilidade não se restringe ao executável, já que muitas das APIs são também específicas do sistema operacional. Assim, não basta recompilar para outra plataforma, é preciso reescrever boa parte do código (Figura 1.1).
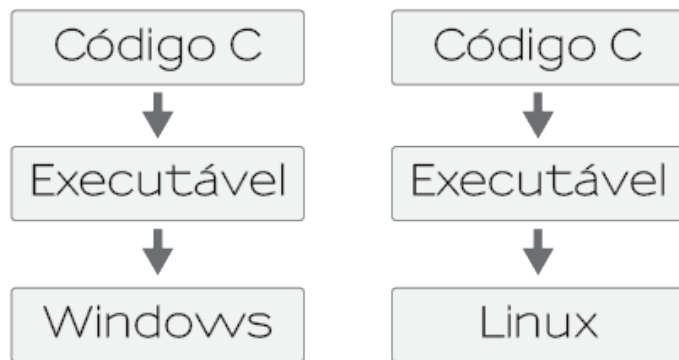
Figura 1.1: Programa em C sendo executado em dois sistemas operacionais diferentes.

Ao usar o conceito de máquina virtual, o Java elimina o problema da portabilidade do código executável. Em vez de gerar um executável específico, como "*Linux PPC*" ou "*Windows i386*", o compilador Java gera um executável para uma máquina genérica, virtual, não física, a JVM.

JVM é uma máquina completa que roda em cima da real. Ela possui suas próprias instruções de máquina (*assembly*) e suas APIs. Seu papel é executar as instruções de máquina genéricas no Sistema Operacional e no hardware específico sob o qual estiver rodando (Figura 1.2).
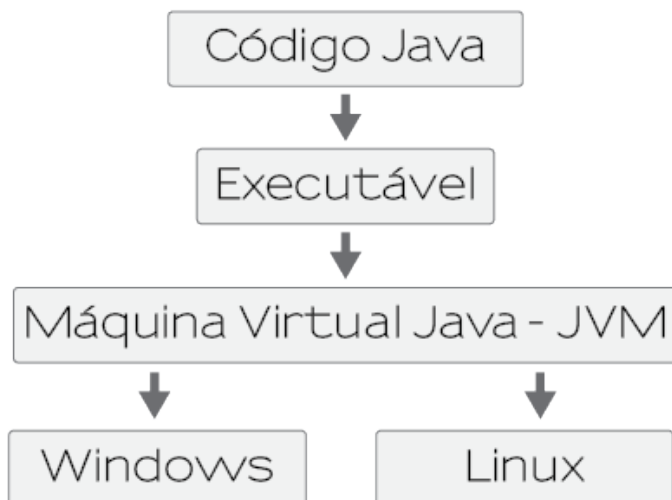
Figura 1.2: Programa em Java sendo executado em dois sistemas operacionais diferentes.

No fim das contas, o problema da portabilidade continua existindo, mas o Java puxa essa responsabilidade para a própria JVM, em vez de deixá-la a cargo do desenvolvedor. É preciso instalar uma JVM específica para o sistema operacional e o hardware que se vai usar. Mas, feito isso, o desenvolvimento do aplicativo é totalmente portável.

É importante perceber que, por causa deste conceito da JVM, o usuário final, que deseja apenas rodar o programa (não desenvolvê-lo), não pode simplesmente executá-lo. Tanto os desenvolvedores quanto os usuários precisam da JVM. Mas os primeiros, além da JVM, precisam do compilador e de outras ferramentas.

Por esta razão, o Java possui duas distribuições diferentes: o **JDK** (**Java Development Kit**), que é focado no desenvolvedor e traz, além da VM, compilador e outras ferramentas úteis; e o **JRE** (**Java Runtime Environment**), que traz apenas o necessário para se executar um aplicativo Java (a VM e as APIs), voltado ao usuário final.

Um ponto fundamental a respeito da JVM é que ela é uma especificação, dife-

rente de muitos outros produtos e linguagens. Quando você comprava o Microsoft Visual Basic 6.0, só havia uma opção: o da Microsoft.

A JVM, assim como a linguagem, possui especificações muito bem definidas e abertas: a *Java Virtual Machine Specification* [113] e a *Java Language Specification.* [85] Em outras palavras, pode-se tomar a liberdade de escrever um compilador e uma máquina virtual para o Java. E é isto o que acontece: não existe apenas a JVM HotSpot da Sun/Oracle, mas também a J9 da IBM, a JRockit da Oracle/BEA, a Apple JVM, entre outras. Ficamos independentes até de fabricante, o que é muito importante para grandes empresas que já sofreram muito com a falta de alternativas para tecnologias adotadas antigamente.

Essa variedade de JVMs também dá competitividade ao mercado. Cada uma das empresas fabricantes tenta melhorar sua implementação, seja aperfeiçoando o JIT *compiler*, fazendo *tweaks* ou mudando o gerenciamento de memória. Isto também traz tranquilidade para as empresas que usam a tecnologia. Se algum dia o preço da sua fornecedora estiver muito alto, ou ela não atingir mais seus requisitos mínimos de performance, é possível trocar de implementação. Temos a garantia de que isso funcionará, pois uma JVM, para ganhar este nome, tem de passar por uma bateria de testes da Sun, garantindo compatibilidade com as especificações.

A plataforma Microsoft .NET também é uma especificação e, por este motivo, o grupo Mono pode implementar uma versão para o Linux.

Os bytecodes Java

A JVM é, então, o ponto-chave para a portabilidade e a performance da plataforma Java. Como vimos, ela executa instruções genéricas compiladas a partir do nosso código, traduzindo-as para as instruções específicas do sistema operacional e do hardware utilizados.

Essas instruções de máquina virtual são os **bytecodes**. São equivalentes aos *mnemônicos* (comandos) do assembly, com a diferença de que seu alvo é uma máquina virtual. O nome *bytecode* vem do fato de que cada *opcode* (instrução) tem o tamanho de um byte e, portanto, a JVM tem capacidade de rodar até 256 bytecodes diferentes (embora o Java 7 possua apenas 205). Isto faz dela uma máquina teoricamente simples de se implementar e de rodar até em dispositivos com pouca capacidade.

Podemos, inclusive, visualizar esses bytecodes. O download do JDK traz uma ferramenta, o **javap,** capaz de mostrar de maneira mais legível o bytecode contido

em um arquivo *.class* compilado. O bytecode será mostrado através dos nomes das instruções, e não de bytes puros.

O código a seguir ilustra um exemplo de classe Java:

```
public class Ano {
    private final int valor;
    public Ano(int valor) {
        this.valor = valor;
    }
    public int getValor() {
        return valor;
    }
}
```

E a execução da linha do comando `javap -c Ano` exibe o conjunto de instruções que formam a classe `Ano`:

```
public class Ano extends java.lang.Object{
public Ano(int);
Code:
0: aload_0
1: invokespecial #10; //Method java/lang/Object.''<init>'':()V
4: aload_0
5: iload_1
6: putfield #13; //Field valor:I
9: return

public int getValor();
Code:
0: aload_0
1: getfield #13; //Field valor:I
4: ireturn
}
```

Embora a linguagem Java tenha um papel essencial na plataforma, tê-la como uma executora de bytecodes, e não exatamente de código Java, abre possibilidades interessantes. A plataforma Java tem ido cada vez mais na direção de ser um ambiente de execução multilinguagem, tanto através da *Scripting API* quanto por linguagens que compilam direto para o bytecode. É uma ideia presente, por exemplo, no .NET desde o início, com sua *Common Language Runtime*(CLR) executando diversas linguagens diferentes.

A linguagem Scala mostra esta força do bytecode para a plataforma Java. Para a JVM, o que interessa são os bytecodes, e não a partir de qual linguagem eles foram gerados. Em Scala, teríamos o seguinte código fonte, que possui a mesma funcionalidade que o código Java mostrado anteriormente:

```
class Ano(val valor:Int)
```

Ao compilarmos a classe Scala anterior com o compilador `scalac`, os bytecodes gerados serão muito próximos dos vistos na classe Java. É possível observar isso rodando o comando `javap` novamente. A única grande mudança é que o getter em Scala chama-se apenas `valor()` e não `getValor()` como em Java. Veremos mais sobre outras linguagens rodando em cima da JVM neste capítulo. Além disso, muitos frameworks e servidores de aplicação geram bytecode dinamicamente durante a execução da aplicação.

A JVM é, portanto, um poderoso executor de bytecodes, e não interessa de onde eles vêm, independentemente de onde estiver rodando. É por este motivo que muitos afirmam que a linguagem Java não é o componente mais importante da plataforma, mas, sim, a JVM e o bytecode.

1.2 Especificações ajudam ou atrapalham?

A Plataforma Java é bastante lembrada por suas características de abertura, portabilidade e até liberdade. Há o aspecto técnico, com independência de plataforma e portabilidade de sistema operacional desde o início da plataforma. Há também a comunidade com seus grupos de usuários por todo o mundo, sempre apoiados e incentivados pela Sun/Oracle. Há ainda o mundo open source, que abraçou o Java com diversos projetos cruciais para o ecossistema da plataforma, muitos deles presentes no dia a dia da maioria dos desenvolvedores. E há o JCP e suas especificações.

Em 1998, três anos após o nascimento do Java dentro da Sun, foi criado o *Java Community Process* (JCP), como um órgão independente da própria Sun, cujo objetivo era guiar os passos do Java de maneira mais aberta e independente. Hoje, a Oracle detém os direitos da marca Java, mas as decisões sobre os rumos da plataforma não estão apenas em suas mãos. Quem guia os rumos do Java é o JCP, com suas centenas de empresas e desenvolvedores participantes, que ajudam a especificar as novas tecnologias e a decidir novos caminhos. Há um *Comitê Executivo* eleito a cada três anos que comanda o órgão. Grandes empresas, como Google, IBM, HP, RedHat, Nokia, além da Oracle, fazem parte dele, inclusive organizações livres, como a SouJava.

Cada nova funcionalidade ou mudança que se queira realizar na plataforma é proposta ao JCP no formato **JSR**, uma *Java Specification Request*. São documentos que relatam essas propostas, para que possam ser votadas pelos membros do JCP. Depois de aceita uma JSR, é formado um **Expert Group** para especificá-la em detalhes e depois implementá-la. Todos os passos ficam sob a chancela do Comitê Executivo, e não unicamente da Oracle. As JSRs recebem números de identificação e acabam ficando reconhecidas por estes. A JSR 318, por exemplo, especificou o EJB 3.1; a JSR 316 para o Java EE 6; e a JSR 336 para o Java SE 7.

Mas as JSR definem apenas *especificações* de produtos relacionados ao Java, e não implementações. Até há uma implementação de referência feita pelo JCP, mas, no fundo, o papel do órgão é criar grandes documentos que definem como tudo deve funcionar. O que usamos na prática é uma implementação compatível com a especificação oficial. Quando baixamos o JDK da Sun/Oracle, o que vem não é o Java SE, mas, sim, uma implementação feita pela empresa seguindo a especificação. Mas não somos obrigados a usar a HotSpot, implementação da Sun/Oracle. Podemos usar outras, como a J9 da IBM, a JRockit da BEA/Oracle, ou o Harmony da Apache, e, ainda, JVMs mais específicas para certos sabores do Unix, dispositivos móveis e hardwares mais particulares.

Por este modelo de abertura de especificações, o Java garante a todos um grau de independência do fabricante. É possível usar produtos de diversos e diferentes fabricantes (IBM, Oracle, JBoss) e trocá-los com certa facilidade na hora que for necessário. As diferenças entre os produtos de diferentes fabricantes estão no suporte oferecido, documentação, facilidades extras de gerenciamento, otimizações, e até extensões proprietárias. Sobre este último ponto vale uma ressalva: praticamente todo fabricante estende a especificação oficial com APIs proprietárias, seja com o objetivo de tapar lacunas na especificação, seja para ter algum destaque a mais em relação a seus concorrentes. Não há problema algum nisso, mas é preciso ter consciência de que, ao usar um recurso proprietário, é possível perder portabilidade e independência do fabricante.

Há muitas críticas, contudo, ao JCP e ao processo de especificação do Java no geral. Fala-se da demora das novas especificações e que o JCP adota apenas aspectos mais básicos das tecnologias, para agradar a todos, travando assim a inovação. Mas, afinal, especificações ajudam ou atrapalham?

Quando uma tecnologia se estabelece como uma das principais em sua área, sua evolução torna-se invariavelmente mais lenta. Novas ideias que surgem no mercado tentam atacar os pontos negativos que a tecnologia padrão possui, e o processo natu-

ral é que uma dessas se torne o novo padrão dentro de algum tempo. Um motivo para a tecnologia padrão não adotar essas inovações é seu sucesso se dever aos resultados já obtidos anteriormente no mercado. Mas problemas e soluções são temporais, dependem da realidade no momento em que são estudados.

A linguagem Java vive uma situação similar. Desenvolvida para resolver determinados problemas, as barreiras que apresenta são muitas vezes superadas pela mistura com outras linguagens dentro da plataforma Java. Um dos grandes conflitos da linguagem é garantir compatibilidade com versões anteriores. É um fator importante para dar segurança aos usuários e garantir larga adoção, e, ao mesmo tempo, um freio na quantidade aceita de inovação e evolução.

Na plataforma .NET há um histórico de menor rigidez em relação à compatibilidade. A linguagem C# evoluiu mais rapidamente, removendo decisões hoje consideradas ruins e incorporando outras consideradas interessantes, como lambdas e *Extension Methods.*[36] Ao mesmo tempo, a quebra de compatibilidade causa descontentamento de diversos clientes. Desenvolvedores precisam de novos treinamentos, o Visual Studio é alterado e diversas bibliotecas deixam de funcionar. Por isso, a partir do .NET 2.0 não houve mais quebras de compatibilidade, apenas novas bibliotecas e funcionalidades.

No caso da plataforma Java, o JCP utiliza seu comitê para definir os rumos das especificações. É uma abordagem conhecida como *Design by Committee*, visando colocar as especificações nas mãos de mais de uma empresa. Ao mesmo tempo, diversas inovações aconteceram e continuam a surgir, em produtos proprietários e open source que, com o passar do tempo e a adoção pelo mercado, acabam sendo padronizados através do JCP. Sendo assim, especificações são comumente criadas, ou para gerar inovações, ou para padronizar aquilo que o mercado inovou e adotou como boas tecnologias.

Porém, o processo introduzido pelo comitê implica maior lentidão na inovação. Se há quebra de compatibilidade, dificilmente a mudança será adotada. Uma funcionalidade pode ser tecnicamente excelente para determinados usuários, mas, se for contra os interesses de um dos membros do comitê, poderá ser deixada de fora. Se não há consenso, não entra, ou, pior, estende-se a discussão por muito tempo.

Eventualmente, extensões à especificação ficam populares, como a *Criteria API* do Hibernate, que não fez parte da primeira versão da JPA (JSR-220). Caso as outras implementações da especificação precisem de muito esforço para suportar uma nova funcionalidade, os membros da especificação podem criar barreiras e a funcionalidade pode até mesmo não entrar para a JSR, ou entrar com diversas mudanças que

não necessariamente fazem sentido do ponto de vista tecnológico. O processo burocrático e o tempo gasto pelos desenvolvedores adaptando a implementação podem defasar as especificações (Figura 1.3).
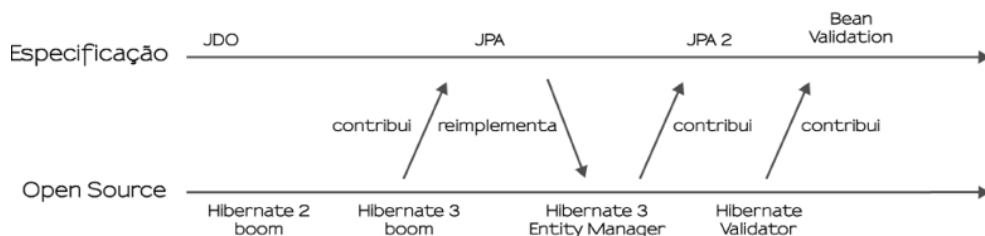
Figura 1.3: Como a inovação no mundo open source está relacionada às especificações.

Mas existem diversos motivos que tornam as especificações atraentes. A mais frequentemente citada é a opção de mudar a implementação, minimizando o efeito de *vendor lock-in*. Porém, na prática, é raro trocarmos a implementação de uma especificação, a não ser talvez considerando diferentes ambientes, como ir do desenvolvimento para a produção.

Apesar de tal vantagem, muitas vezes uma implementação específica acaba por dominar o mercado, como no caso do Hibernate. É raro encontrar situações em que equipes trocaram o Hibernate por outra implementação de JPA, mas é comum outras implementações serem trocadas pelo Hibernate. Nesse sentido, se o projeto já usa Hibernate, a especificação pode mais limitar que engradecer o projeto. Além disso, otimizações costumam ser feitas para implementações específicas.

Ao adotar uma especificação, o programador e o arquiteto devem levar em consideração as perdas, devido à abstração, e os ganhos de um padrão, que facilite a entrada de novos desenvolvedores e desacopla o sucesso de um projeto de um único *vendor*.

Uma grande vantagem das especificações é a certeza de que conhecê-la permitirá ao desenvolvedor trabalhar em projetos que tenham diferentes implementações. Claro que haverá diferenças a serem aprendidas, mas ter a especificação diminui bastante esse aprendizado.

Outro ponto muitas vezes citado é que as especificações trazem certas garantias de futuro e continuidade para aquela tecnologia, deixando o projeto livre até de de-

terminado fabricante caso ele pare de suportar seu produto ou mude sua política comercial. Isto não significa que projetos open source e tecnologias proprietárias não tenham essas garantias, mas adotar uma especificação pode trazer ainda mais garantias de futuro quando isso for muito necessário. Porém, há casos também de especificações praticamente abandonadas, como a JDO, que é muito pouco usada e já superada pela JPA.

Como qualquer escolha, desenvolver voltado à especificação ou usar tecnologias proprietárias é uma decisão a ser tomada a partir das vantagens e desvantagens de cada abordagem. A plataforma Java, diferente de muitas outras, tem essa possibilidade de escolha disponível em suas diversas ramificações. Quando for preciso uma especificação de alguma tecnologia, provavelmente a plataforma Java terá algo disponível. E, quando for conveniente a liberdade e inovação das tecnologias proprietárias, certamente também encontraremos algo.

1.3 Use a linguagem certa para cada problema

Com o passar do tempo e o amadurecimento da plataforma Java, percebeu-se que a linguagem ajudava em muitos pontos, mas também impedia um desenvolvimento mais acelerado em partes específicas.

Os primeiros indícios da necessidade de outras funcionalidades na linguagem Java são antigos. Algumas delas foram supridas com a disseminação de técnicas, como proxies dinâmicos, em vez dos stubs pré-compilados encontrados em RMI e EJB 1.x e 2.x. Ou, ainda, as anotações Java para diminuir as configurações XML, antes comuns em frameworks como o XDoclet e o Spring.

Outras necessidades foram resolvidas com a manipulação de bytecodes em tempo de execução com a explosão na adoção de bibliotecas com esse fim, como ASM ou Javassist. Há ainda linguagens como Groovy e outras foram desenvolvidas para facilitar a criação de código mais adequado à tipagem dinâmica.

Com isso, outras linguagens, até então consideradas secundárias pelo mercado, começaram a tomar força por possuírem maneiras diferenciadas de resolver os mesmos problemas, mostrando-se mais produtivas em determinadas situações.

Para aplicações Web em Java, por exemplo, é comum a adoção de outras linguagens. Grande parte do tempo, utiliza-se CSS para definir estilos, HTML para páginas, JavaScript para código que será rodado no cliente, SQL para bancos de dados, XMLs de configuração e *expression language* (*EL*) nos JSPs. Mas, se já usamos a linguagem certa para a tarefa certa, por que não aproveitar esta prática em toda a aplicação?

Por que não acrescentar mais uma linguagem se esta resolver o mesmo problema de maneira mais simples e elegante?

A plataforma .NET já foi criada tendo como um de seus objetivo suportar diversas linguagens. A Sun percebeu posteriormente que não apenas a linguagem Java poderia se aproveitar das vantagens da plataforma: o excelente compilador JIT, algumas das melhores implementações de Garbage Collector existentes, a portabilidade e a quantidade de bibliotecas já consolidadas no mercado e prontas para ser usadas.

Hoje, a JVM é capaz de interpretar e compilar código escrito em diversas linguagens. Com o Rhino, código JavaScript pode ser executado dentro da JVM. Assim como código Ruby pode ser executado com JRuby, Python com Jython e PHP com Quercus. Existem linguagens criadas especificamente para rodar sobre a JVM, como Groovy, Beanshell, Scala e Clojure. O alemão Robert Tolksdorf mantém uma lista da maioria das linguagens suportadas pela JVM em seu site,[204] onde é possível encontrar até mesmo implementações da linguagem Basic.

Muitos perguntam qual seria a utilidade de executar, por exemplo, código JavaScript na JVM. A resposta está nas características da linguagem, afinal, a plataforma será a mesma. Apesar de todo o preconceito que existe sobre ela, JavaScript vem se tornando uma linguagem de primeira linha, que é adotada em outros pontos de nossa aplicação, e não apenas no contato final com o cliente. O artigo "*A re-introduction to JavaScript*" mostra como a linguagem vai além dos tutoriais simples que encontramos na Internet.[215] E a adoção de plataformas servidoras, como o Node.js, mostram como JavaScript também pode ser uma linguagem de uso geral.

Um exemplo simples de uso de JavaScript é para evitar a duplicação de código; se já existe uma validação de CPF do cliente feita via JavaScript, podemos reutilizar esse pedaço de código no servidor, garantindo o funcionamento do sistema mesmo no caso de JavaScript estar desabilitado no cliente. O código continua sendo executado dentro da JVM com todas as otimizações ligadas à compilação sob demanda do mesmo. Não é à toa que as novas versões do Hibernate Validator permitem que sua validação *server side* seja em JavaScript.

É comum encontrar aplicações Java rodando Ruby durante o processo de build, através de ferramentas como Rake e Cucumber, para, por exemplo, efetuar os testes *end-to-end* através de um browser. O Rails rodando sob a JRuby é cada vez mais adotado.

O Grails, um framework baseado nos conceitos do Rails, mas levado à plataforma Java através da linguagem Groovy, também é adotado para a criação de projetos Web em geral. Essa linguagem também pode ser utilizada juntamente com

Beanshell ou Ruby para configurar frameworks como o Spring, que já não está mais só limitado a XML.

Em arquiteturas modernas, é comum encontrar aplicações Web desenvolvidas sob a plataforma Java, por exemplo, no cloud do Google App Engine, escritas em Java e em Ruby, interagindo através de JRuby e testada via Ruby e JSpec. O desenvolvedor precisa dominar cada vez mais linguagens para tirar proveito de todas elas. Há o contraponto na manutenibilidade do código: mais linguagens exigem mais trabalho, e é necessário medir muito bem esse *trade-off*.

A tendência na mescla de linguagens é tão forte que a Sun criou o projeto *MLVM* (*MultiLanguage Virtual Machine*), também conhecido como *Da Vinci Machine*. O projeto busca viabilizar um conjunto de especificações para melhorar a execução de código escrito em outras linguagens, especialmente as linguagens dinâmicas. Existe uma iniciativa parecida na plataforma .NET, o **D**ynamic **L**anguage **R**untime – **DLR**.

Um dos principais avanços trazidos pelo projeto *Da Vinci Machine* é um novo bytecode chamado `invokedynamic`, uma das principais mudanças no Java SE 7. O `invokedynamic` permite que métodos não definidos em tempo de compilação possam ser acessados através de código Java puro, facilitando a mistura com outras linguagens que trouxeram objetos com características dinâmicas para dentro da *Virtual Machine*, como o JRuby. É um momento importante na direção do Java como plataforma multilinguagem. É a primeira vez que um novo bytecode é adicionado à plataforma para não ser usado pela linguagem Java em si, mas por outras linguagens.

Ainda seguindo esse caminho da linguagem correta no instante adequado, surgiu um forte movimento que busca o bom uso de linguagens apropriadas ao domínio a ser atacado, as *Domain Specific Languages*. Existem diversos tipos, características e maneiras de criá-las, sempre com o intuito de trazer para o código uma legibilidade que seja mais natural.

Linguagens funcionais

Em 2009, Steve Vinoski chamou a atenção para a retomada da popularidade do uso de linguagens funcionais na Web,[207] causada principalmente pela facilidade de se trabalhar com concorrência e expressividade.

Operações comuns, como tirar uma média de preço de uma `List<Produto>`, vão consumir algumas linhas de código em Java, mesmo com o auxílio de bibliotecas externas que possuam `Predicates`. Com o funcional, como em Scala, um simples `media(produtos)(_.preco)` faz esse trabalho, e pode ser reaproveitado para calcular a média de tempo gasto de uma `List<Processos>`, dado que temos inúmeras

primitivas funcionais, como o `foldLeft` (conhecido como `accumulate` ou `reduce` em outras linguagens):

```
def media[T](itens:List[T])(valorador: T => Double) =
itens.foldLeft(0.0)(_ + valorador(_)) / itens.size
```

Quando Java quebrou a barreira de que máquinas virtuais seriam necessariamente lentas, ajudou a desconstruir também o mito similar a linguagens funcionais, que existem em versões interpretadas quanto compiladas.

Por outro lado, com a desaceleração do aumento da potência de um processador, o mercado voltou-se para o lado dos computadores multi-core e linguagens funcionais mais puras, fortemente embasadas na imutabilidade, que podem se beneficiar de tais processadores de maneira mais simples e eficiente, sem uso manual de locks e delimitações de regiões críticas. A facilidade trazida pelas closures, a partir do Java SE 8,[186] é também um diferencial a ser considerado, diminuindo drasticamente o número de classes anônimas, tornando o código mais legível e menos verboso.

Com isso, padrões que são mais simples de implementar em linguagens funcionais podem ser mais complicados em outros paradigmas, e vice-versa.

Podemos escolher a melhor linguagem para cada tarefa, reaproveitando toda a biblioteca já existente em Java, e comunicando entre elas de maneira transparente.

Capítulo 2

Java Virtual Machine

Com a JVM no centro da Plataforma Java, conhecer seu funcionamento interno é essencial para qualquer aplicação Java. Muitas vezes, deixamos de lado ajustes importantes de Garbage Collector, não entendemos as implicações do JIT no nosso design ou enfrentamos problemas práticos com ClassLoaders.

Dentre os diversos tópicos associados à JVM, destacamos alguns que julgamos vitais para todo desenvolvedor Java.

2.1 Princípios de garbage collection

Durante muito tempo, uma das maiores dificuldades na hora de programar era o gerenciamento de memória. Os desenvolvedores eram responsáveis pela sua alocação e liberação manualmente, o que levava a muitos erros e *memory leaks*. Hoje, em todas as plataformas modernas, Java inclusive, temos gerenciamento de memória automático através de algoritmos de coleta de lixo.

O **Garbage Collector** (GC) é um dos principais componentes da JVM e responsável pela liberação da memória que não esteja mais sendo utilizada. Quando a

aplicação libera todas as referências para algum objeto, este passa a ser considerado lixo e pode ser coletado pelo GC a qualquer momento. Mas não conseguimos determinar o momento exato em que essa coleta ocorrerá; isto depende totalmente do algoritmo do garbage collector. Em geral, o GC não fará coletas para cada objeto liberado; ele deixará o lixo acumular um pouco para fazer coletas maiores, de maneira a otimizar o tempo gasto. Essa abordagem, muitas vezes, é bem mais eficiente, além de evitar a fragmentação da memória, que poderia aparecer no caso de um programa que aloque e libere a memória de maneira ingênua.[39]

Mas, como é realizada essa coleta exatamente? Vimos que ela não é feita logo que um objeto fica disponível, mas, sim, de tempos em tempos, tentando maximizar a eficiência. Em geral, a primeira ideia que aparece ao se pensar em GC é que ele fica varrendo a memória periodicamente e libera aqueles objetos que estão sem referência. Esta é a base de um conhecido algoritmo de GC chamado **Marck-And-Sweep**,[159] constituído de duas fases: na primeira, o GC percorre a memória e marca (*mark*) todos os objetos acessíveis pelas threads atuais; na segunda, varre (*sweep*) novamente a memória, coletando os objetos que não foram marcados na fase anterior.

Esse algoritmo envelheceu, da mesma forma que o ingênuo *reference counting*. Estudos extensivos com várias aplicações e seus comportamentos em tempo de execução ajudaram a formar premissas essenciais para algoritmos modernos de GC. A mais importante delas é a **hipótese das gerações**.[206]

Segundo esta hipótese, geralmente 95% dos objetos criados durante a execução do programa têm vida extremamente curta, isto é, são rapidamente descartados. É o que artigos acadêmicos chamam de alto índice de *mortalidade infantil* entre os objetos. A hipótese das gerações ainda diz que os objetos sobreviventes costumam ter vida longa. Com base nessas observações, chegou-se ao que hoje é conhecido como o algoritmo **generational copying**, usado como base na maioria das máquinas virtuais.[137] [160]

É simples observar esse padrão geracional em muitos programas escritos em Java, quando objetos são criados dentro de um método. Assim que o método termina, alguns objetos que foram criados lá ficam sem referências e se tornam elegíveis à coleta de lixo, isto é, eles sobreviveram apenas durante a execução do método e tiveram vida curta.

Mesmo métodos curtos e simples, como `toString`, acabam gerando objetos intermediários que rapidamente não serão mais referenciados:

```
public String toString() {
    return ‘‘[ contatos: ‘‘ + listaDeContatos + ‘‘]’’;
```

```
}
```

Aqui, durante a concatenação das três partes da `String`, um `StringBuilder` será utilizado, e o mesmo vai ocorrer para a invocação implícita do `toString` da coleção `listaDeContatos`, que gera uma `String` a partir de outro `StringBuilder`. Todos esses objetos podem ser rapidamente descartados, e talvez o próprio retorno do método será referenciado apenas por um breve instante.

No *generational copying*, a memória é dividida em gerações, que geralmente são duas *young generation* e *old generation* (Figura 2.1). A geração nova é tipicamente menor que a velha; a JVM HotSpot, por exemplo, está programada para ocupar, por padrão, até um terço do espaço total. Todo novo objeto é alocado nesta parte e, quando ela se encher, o GC realizará seu trabalho em busca de sobreviventes. O truque está justamente no fato de que o GC, neste caso, varre apenas a geração jovem, que é bem pequena, sem ter de paralisar a JVM (*stop-the-world*) por muito tempo.

Os objetos que sobrevivem à coleta são, então, copiados para a geração seguinte, e todo o espaço da geração nova é considerado disponível novamente. Esse processo de cópia de objetos sobreviventes é que dá nome ao algoritmo.

Figura 2.1: Divisão das gerações no heap.

Pode-se pensar que o *generational copying* não é tão bom, pois, além de liberar objetos da memória, gasta tempo copiando os sobreviventes. Mas seu grande trunfo é que ele age nos objetos sobreviventes, e não nos descartados, como faria um algoritmo tradicional. No descarte, os objetos não são verdadeiramente apagados da memória; o GC apenas marca a memória como disponível. Segundo a hipótese das gerações, portanto, o *generational copying* realizará cópia em apenas 5% dos objetos, os que têm vida mais longa. E, embora uma cópia seja relativamente custosa, copiar apenas os poucos sobreviventes é mais rápido que liberar, um por um, os diversos objetos mortos.

Novos objetos são alocados na young e, assim que ela estiver lotada, é efetuado o chamado **minor collect**. É neste passo que os objetos sobreviventes são copiados para a old, e todo o espaço da young torna-se disponível novamente para aloca-

ção. Com menor frequência, são executados os **major collects**, que também coletam na geração old usando o *mark-and-sweep*. *Major collects* são também chamados **FullGC**, e costumam demorar bem mais, já que varrem toda a memória, chegando a travar a aplicação nos algoritmos mais tradicionais (não paralelos).

É possível fazer uma observação sucinta do comportamento do GC mesmo sem um profiler, bastando usar a opção `-verbose:gc` ao iniciar a JVM. O log de execução do GC mostra a frequência das chamadas aos *minor* e *major collects*, bem como a eficiência de cada chamada (quanto de memória foi liberado) e o tempo gasto. É importante observar esses valores para perceber se o programa não está gastando muito tempo nos GCs, ou se as coletas estão sendo ineficientes. Um gargalo possível de ser encontrado é a geração young muito pequena, fazendo com que muitos objetos sejam copiados para a old para, logo em seguida, serem coletados em uma *major collect*, prejudicando bastante a performance geral do GC e a eficiência dos *minor collects*.

Um mito muito comum é o de que estressamos o GC se criarmos muitos objetos. Na verdade, como os algoritmos estão adaptados segundo a hipótese das gerações, o melhor são muitos pequenos objetos que logo se tornam desnecessários, do que poucos que demoram para sair da memória. Em alguns casos, até o tamanho do objeto pode influenciar; na JRockit, por exemplo, objetos grandes são alocados direto na ::old generation:;, logo não participam da cópia geracional.

A melhor técnica que um desenvolvedor pode utilizar é encaixar a demanda de memória da sua aplicação na hipótese das gerações e nas boas práticas de orientação a objetos, criando objetos pequenos e encapsulados de acordo com sua necessidade. Se o custo de criação do objeto não for grande, segurar suas referências ou fazer caches acaba sendo pior. Obviamente, isso exclui casos em que o custo de criação é grande, como um laço de concatenação de `String` através do operador +; nesse caso, é melhor usar `StringBuilders` ou `StringBuffers`.

Um efeito colateral interessante do algoritmo de cópia de objetos é a **compactação** da memória. Algoritmos ingênuos de GC costumam causar grande fragmentação, porque apenas removem os objetos não mais usados, e os sobreviventes acabam espalhados e cercados de áreas vazias. O *generational copying* copia os objetos sobreviventes para outra geração de forma agrupada, e a memória da geração anterior é liberada em um grande e único bloco, sem fragmentação. Fora isso, outras estratégias de compactação de memória ainda podem ser usadas pela JVM, inclusive na *old generation*. É importante notar que isso só é possível por causa do modelo de memória do Java, que abstrai totalmente do programa a forma como os ponteiros

e endereços de memória são usados. É possível mudar objetos de lugar a qualquer momento, e a VM precisa apenas atualizar seus ponteiros internos, o que seria muito difícil de realizar em um ambiente com acesso direto a ponteiros de memória.

Explorando o gerenciamento de memória nas JVMs

A área total usada para armazenar os objetos na JVM é chamada heap. O tamanho do heap de memória da máquina virtual é controlado pelas opções `-Xms` e `-Xmx`. A primeira especifica o tamanho inicial do heap, e a segunda, o tamanho máximo. Inicialmente, a JVM aloca no sistema operacional a quantidade Xms de memória de uma vez, e essa memória nunca é devolvida para o sistema. A alocação de memória para os objetos Java é resolvida dentro da própria JVM, e não no sistema operacional. Conforme mais memória é necessária, a JVM aloca em grandes blocos até o máximo do Xmx (se precisar de mais que isso, um `OutOfMemoryError` é lançado). É muito comum rodar a máquina virtual com valores iguais de Xms e Xmx, fazendo com que a VM aloque memória no sistema operacional apenas no início, deixando de depender do comportamento específico do SO.

Conhecer essas e outras opções do garbage collector da sua JVM pode impactar bastante na performance de uma aplicação. Considere o código a seguir:

```
for (int i = 0; i < 100; i++) {
    List<Object> lista = new ArrayList<Object>();
    for (int j = 0; j < 300000; j++) {
        lista.add(new Object());
    }
}
```

Rodando um programa com esse código no main na JVM 6 HotSpot, usando o `-client` padrão com 64m de Xms e Xmx, ao habilitar o `-verbose:gc` obtemos uma saída como esta:

```
[GC 25066K->24710K(62848K), 0.0850880 secs]
[GC 28201K(62848K), 0.0079745 secs]
[GC 39883K->31344K(62848K), 0.0949824 secs]
[GC 46580K->37787K(62848K), 0.0950039 secs]
[Full GC 53044K->9816K(62848K), 0.1182727 secs]
....
```

Os valores mostrados em cada linha correspondem a: memória em uso antes da coleta, memória depois, total de memória da JVM e o tempo gasto na coleta. A saída

real é bem maior, mas praticamente repete os mesmos padrões. É possível perceber que os *minor* GC são muito pouco eficientes, liberam pouca memória, ao contrário dos *Full GC* que parecem liberar muita. Isso indica fortemente que estamos indo contra a hipótese das gerações. Os objetos não estão sendo liberados enquanto ainda jovens, nem sendo copiados para a geração velha, de onde, depois de um tempo, serão liberados. Neste nosso caso, o algoritmo de GC baseado em cópia de objetos é um imenso gargalo de performance.

Propositadamente, o código de teste segura referências por um tempo que estressa o GC. Em uma aplicação real, o ideal seria alterar o código para que se adapte melhor à hipótese das gerações. Mas, quando o comportamento da sua aplicação é diferente do habitual, basta alterar algumas configurações do garbage collector para obter algum ganho.

Em nossa máquina de testes, sem alterar o tamanho total do heap, ao aumentar o da young generation usando a opção `-XX:NewRatio=1` (o padrão é 2 na HotSpot), temos um ganho de mais de 50% na performance geral da aplicação. Essa opção muda a proporção entre os tamanhos da *young generation* e da *old*. Por padrão, 2/3 serão *old* e 1/3 será *young*; ao mudar o valor para 1, teremos 1/2 para cada área. Claro que esse ganho depende muito do programa e da versão da JVM, mas, ao olhar agora para a saída do programa, é possível observar como uma pequena mudança impacta bastante na eficiência do GC:

```
[GC 49587K->25594K(61440K), 0.0199301 secs]
[GC 43663K->26292K(61440K), 0.0685206 secs]
[GC 47193K->23213K(61440K), 0.0212459 secs]
[GC 39296K->21963K(61440K), 0.0606901 secs]
[GC 45913K->21963K(61440K), 0.0215563 secs]
...
```

Nos novos testes, o *Full GC* nem chega a ser executado, porque os próprios *minor gc* são suficientes para dar conta dos objetos. E, como o algoritmo de GC é baseado em cópias, poucos objetos são copiados para a *old generation*, influenciando bastante na performance do programa. **Saiba adequar sua aplicação à hipótese das gerações, seja adequando seu código, seja conhecendo as melhores configurações de execução da sua JVM.**

Pensando em como usufruir dessas características do GC, o que é melhor para uma aplicação: usar objetos grandes, caches gigantes de objetos e abusar de atributos estáticos; ou escrever diversos pequenos objetos criados e liberados a todo momento? Segurar objetos na memória estressa demais o GC baseado em cópias

de objetos. A boa prática de Orientação a Objetos já diz que devemos criar pequenos objetos encapsulados, sem muitos dados estáticos, originando instâncias sem nos preocuparmos com caches e outras alegadas otimizações. Se seguirmos a boa prática, o GC também funcionará melhor. A JVM, de um modo geral, foi escrita e adaptada ao longo de sua vida para suportar o desenvolvedor nas boas práticas.

Outro ponto particular da JVM HotSpot da Oracle/Sun (e suas derivadas) é a chamada *permanent generation*, ou **PermGen** (Figura 2.2). É um espaço de memória contado fora do heap (além do Xms/Xmx portanto), onde são alocados objetos internos da VM e objetos Class, Method, além do **pool de strings**. Ao contrário do que o nome parece indicar, esse espaço de memória também é coletado (durante os *FullGC*), mas costuma trazer grandes dores de cabeça, como os conhecidos `OutOfMemoryError`, acusando o fim do *PermGen space*. A primeira dificuldade aqui é que, por não fazer parte do Xms/Xmx, o *PermGen* confunde o desenvolvedor, que não entende como o programa pode consumir mais memória do que a definida no `-Xmx`. Para especificar o tamanho do PermGen, usa-se outra opção, a `-XX:MaxPermSize`.
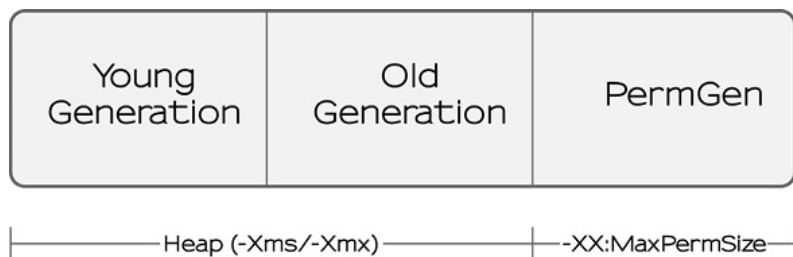
Figura 2.2: Divisão da memória na JVM da Sun.

Mas os problemas com estouro do *PermGen* são difíceis de diagnosticar, justamente porque não se trata de objetos da aplicação. Na maioria dos casos, está ligado a uma quantidade exagerada de classes que são carregadas na memória, estourando o tamanho do *PermGen*. Um exemplo conhecido, que acontecia antigamente, era usar o Eclipse com muitos plugins (sabidamente o WTP) nas configurações padrões da JVM. Por causa da arquitetura de plugins bem organizada e encapsulada do Eclipse, muitas classes eram carregadas e a memória acabava.

Um outro problema bem mais corriqueiro são os estouros do *PermGen* ao se fazer muitos *hot deploys* nos servidores de aplicação. Por vários motivos possíveis (como veremos durante o tópico de ClassLoaders), o servidor não consegue liberar

as classes do contexto ao destruí- lo. Uma nova versão da aplicação é então carregada, mas as classes antigas continuam na memória. É apenas uma questão de tempo para esse vazamento de memória estourar o espaço do *PermGen*.

Algoritmos de coleta de lixo

Aplicações de grande porte podem sofrer com o GC. É fundamental conhecer algumas das principais opções que sua JVM possibilita. No caso da HotSpot, a JVM mais utilizada, diversos algoritmos diferentes estão disponíveis para uso.[135][129] Por padrão, é usado o **Serial Collector**, que é bastante rápido, mas usa apenas uma thread. Em máquinas com um ou dois processadores, ele costuma ser uma boa escolha. Mas em servidores mais complexos, com vários processadores, pode desperdiçar recursos. Isto porque ele é um algoritmo *stop-the-world*, o que significa que todo processamento da aplicação deve ser interrompido enquanto o GC trabalha, para evitar modificações na memória durante a coleta. Mas, por ser serial, implica que, se uma máquina tiver muitos processadores, apenas um poderá ser usado pelo GC, enquanto todos os demais ficam ociosos, pois não podem executar nenhuma outra tarefa enquanto o GC roda. Isto é um desperdício de recursos, e um gargalo em máquinas com vários processadores.

É possível então, na HotSpot, habilitar diversos outros algoritmos de GC e até usar combinações deles. O **Parallel Collector** (habilitado com `-XX:+UseParallelGC`) consegue rodar *minor collects* em paralelo. Ele também é *stop-the-world*, mas aproveita os vários processadores, podendo executar o GC. Note que, por causa disso, o algoritmo paralelo acaba demorando um tempo total maior que o serial, em razão do custo de sincronizar o acesso à memória sendo coletada (`synchronized`, semáforos, mutexes, ...). Mas, como estamos falando de máquinas com muitos processadores, o que acaba acontecendo é que esse tempo é dividido entre os vários processadores, e o efeito é que se usa melhor o hardware e se diminui o tempo de parada para coleta.[135] É possível ainda habilitar um coletor paralelo para a geração old e os major collects, usando `-XX:+UseParallelOldGC`.

Quando o requisito principal é diminuir o tempo de resposta, ou seja, diminuir o tempo que a aplicação fica parada esperando o GC, o melhor pode ser usar o **Concurrent Mark-and-sweep** (CMS), também chamado de *low pause collector* (ativado com `-XX:+UseConcMarkSweepGC`). Esse algoritmo age na geração *old* e consegue fazer a maior parte da análise da memória sem parar o mundo. Ele tem apenas um pequeno pedaço *stop-the-world*, o que acaba tornando o tempo de resposta bem menor. Mas o preço que se paga é que o algoritmo gasta mais processamento total, e

tem custos maiores de controle de concorrência. Pode ser uma boa ideia em máquinas com muitos processadores e com bastante memória a ser coletada.[135] Ao usar o CMS, a geração *young* é coletada com um algoritmo paralelo chamado ParNew e é possível até desabilitá-lo e usar o Serial.

A última novidade em relação a algoritmos GC é o **G1**, desenvolvido pela Sun.[134] É um algoritmo concorrente que tenta trazer um pouco mais de previsibilidade para o GC, uma das grandes críticas de quem prefere gerenciamento manual de memória. Apesar de ser considerado um algoritmo geracional, ele não separa o heap em duas gerações, e sim em muitas pequenas regiões. Dinamicamente, o G1 escolhe algumas regiões para ser coletadas (o que acaba dando o sentido lógico daquelas regiões serem a *young gen*). O interessante é que é possível especificar para o G1 tempo máximo a ser gasto em GC em um determinado intervalo de tempo. O G1, então, tenta escolher um número de regiões para cada coleta que, baseado em execuções anteriores, ele acha que atingirão a meta solicitada. Com isso, espera-se ter um algoritmo de GC mais eficiente e mais previsível, com paradas mais curtas e constantes.

Esses algoritmos mudam muito de uma JVM para outra. Os quatro já citados são das últimas versões da JVM 6 e da JVM 7 da Oracle/Sun. Outras máquinas virtuais possuem outros algoritmos. A JRockit, por exemplo, permite desabilitar o algoritmo geracional e usar um baseado no *mark-and-sweep* com a opção `-Xgc:singlecon` (há outros também parecidos com os algoritmos parallel e concurrent do HotSpot).[43] A Azul Systems, conhecida por sua JVM focada em alta performance, possui em seus produtos, desde o fim de 2010, um GC sem pausas.[196]

A memória pode ser um gargalo para sua aplicação; portanto, conhecer bem seu funcionamento e seus algoritmos pode ser peça-chave para o sucesso de um projeto.

2.2 Não dependa do gerenciamento de memória

Como vimos, o *Garbage Collector* é um dos serviços mais importantes da JVM e exige pouco esforço para usá-lo. Conhece-lo, além de seus algoritmos, ajuda no momento de escrever códigos melhores e não ter surpresas. Mas há ainda outras questões relacionadas ao comportamento do GC.

Um dos principais pontos é sua natureza imprevisível. Por ser um serviço da JVM que roda simultaneamente à sua aplicação, não conseguimos controlar explicitamente seu comportamento. Diferente do gerenciamento manual, no qual tanto a alocação quanto a liberação de memória são chamadas explicitamente pelo de-

senvolvedor, em Java e em outras linguagens baseadas na existência de um GC, não definimos os pontos de coleta. Apenas liberamos as referências para os objetos e, em algum momento, a coleta será efetuada.

A não exigência da necessidade de explicitar essa coleta é o ponto positivo do GC, pois tira essa preocupação do desenvolvedor, mas é também sua maior crítica, pela sua imprevisibilidade. Ele pode ficar muito tempo sem rodar no caso de existir muita memória disponível e, de repente, gastar muito tempo em uma única execução, potencialmente paralisando o restante da aplicação. São essas paralisações que os algoritmos *Concurrent* e G1 tentam minimizar.

Alguns desenvolvedores tentam escrever código para que o próprio programa controle alguns aspectos relacionados ao GC e, de um modo geral, isso não é uma boa ideia.

Uma primeira má prática bastante comum é a invocação ao `System.gc()` na esperança de minimizar a imprevisibilidade do GC e aumentar sua frequência de execuções. Essa abordagem apresenta diversos problemas. Em primeiro lugar, o método não garante a execução do GC, como definido em seu JavaDoc:

Invocar o método GC sugere que a Máquina Virtual Java faça um esforço para reciclar objetos não utilizados a fim de liberar a memória que eles ocupam atualmente para rápida reutilização. Quando o controle volta da chamada do método, a Máquina Virtual Java fez o máximo esforço para recuperar espaço de todos os objetos descartados.[42]

Ou seja, a invocação do método é uma sugestão para a execução do GC, que pode ser atendida ou não. Em outras palavras, a invocação ao `System.gc()` não garante resultado algum. Por exemplo, a JVM da Oracle/Sun, por padrão, sempre segue a sugestão, e toda invocação a este método executa um *Full GC*. Na maioria das vezes, essa obediência é ainda pior, uma vez que o *Full GC*, que não deveria ser executado com muita frequência, varre todo o heap, demorando para completar. Além disso, a quantidade de memória liberada será pequena no caso de invocações muito próximas.

Um caso encontrado com frequência aparece quando um processo dentro de um laço consome muita memória, e os desenvolvedores decidem invocar `System.gc()` dentro do loop, acarretando uma enorme perda de performance:

```
public void processa(List<Cliente> clientes, List<Evento> eventos) {
    for(Cliente cliente : clientes) {
        for(Evento evento : eventos) {
            cliente.analisa(evento);
```

```
            System.gc();
        }
    }
}
```

Podemos tentar fixar um intervalo de invocações do GC, como a cada cinco minutos ou ao término da execução de cada método. Mas é quase certo que a VM seria capaz de escolher momentos mais apropriados para execução do GC do que nós. Isso porque, ao contrário da VM, o programa não tem a visão geral da memória durante todo o tempo de execução e, por isso, é incapaz de calcular exatamente o impacto da coleta.

Por fim, mesmo que insistamos em fazer a tal invocação explícita, é importante saber que, em runtime, é possível desabilitar a execução arbitrária do GC, o que será um problema se não controlamos tudo do ambiente de produção. Na HotSpot, por exemplo, a opção `XX:+DisableExplicitGC`, passada para a JVM na linha de comando, desabilita as chamadas a System.gc(), ignorando-as por completo. Aliás, este é o comportamento padrão de muitas outras JVMs.

O método finalize de Object

Não podemos controlar o momento da coleta dos objetos, mas é possível ser dele notificado nesse momento. Um instante antes de remover cada objeto da memória, o *Garbage Collector* invoca seu método finalize, notificando-o da coleta. Este método é definido na classe `Object`, e podemos reescrevê-lo em nossas classes com código próprio a ser executado antes de os objetos serem coletados.

```
public class DAO {
    private Conexao conexao;

    // métodos do DAO

    @Override
    public void finalize() {
        super.finalize();
        this.conexao.fecha();
    }
}
```

Na maioria das vezes, porém, usar este método causa mais dores de cabeça do que resolve problemas. O método pode ser executado muito tempo depois que o

objeto se tornou inacessível, ou até mesmo não ser chamado nunca. Imagine um programa que roda com muita memória, quase não a usa e executa rapidamente; o GC pode não ter jamais a oportunidade de ser executado.

Mesmo quando for executado, seu comportamento é complicado. E se, durante a finalização, recuperarmos uma referência do objeto em outro lugar? O GC não o coleta e, na próxima vez que o objeto estiver disponível, o finalize não será executado novamente. Ou ainda se nossa implementação deste método deixar vazar uma exceção, ela será engolida e a finalização do objeto interrompida. Se estivermos trabalhando com herança, precisamos garantir a invocação do finalize da mãe. Além disso, as finalizações são executadas sequencialmente por uma thread específica, que possui uma fila de objetos finalizáveis em ordem não determinada, o que faz com que uma possível demora na finalização de um objeto atrase a de algum outro importante. Todos esses problemas se acumulam e dificultam a escrita de um método finalize adequado.

De modo geral, evite depender de finalizações, ainda mais se for um código importante e que necessita da garantia de execução. Não use o finalize como único mecanismo de liberação de recursos importantes, como, por exemplo, fechar conexões ou arquivos. Sempre que seu objeto exigir algum processo de finalização, exponha um método close ou dispose para os desenvolvedores chamarem-no explicitamente, garantindo a execução desse código importante. Com o recurso de *try-with-resources* do Java 7, então, é ainda mais fácil garantir que o desenvolvedor chame o close no momento certo.[45]

Um uso possível para o finalize é como segundo mecanismo de finalização, invocando o método principal caso o usuário se esqueça de fazê-lo. É uma segurança a mais, mas não uma garantia de execução - nesse caso, o usuário estava errado, pois o correto seria ter chamado o close explicitamente.

```
public class Leitor {
    private final InputStream input;
    public Leitor(InputStream input) {
        this.input = input;
    }
    // métodos
    public void finalize() {
        close();
    }
    public void close() {
        try {
```

```
            input.close();
        } catch(IOException ex) {
            // logar ou algo diferente
        }
    }
}
```

Nos raros casos em que precisarmos trabalhar com finalizações, o Java disponibiliza um outro mecanismo mais robusto que o `finalize`.[161] No pacote `java.lang.ref`,[40] estão as classes de referências fracas (`WeakReference`, `SoftReference` e `PhantomReference`) e `RefereceQueue`. Com elas, o desenvolvedor consegue controlar sua própria fila de coleta e processar do jeito que desejar as finalizações de seus objetos. Outro uso de referências fracas é permitir a implementação de caches de objetos que nunca estouram a memória. Por não serem referências normais Java (fortes), elas não impedem a coleta do objeto, simplificando a implementação deste tipo de cache. Os detalhes de referências fracas estão fora do escopo desse livro, mas recomendamos seu uso quando for necessário detectar finalizações de objetos.

De um modo geral, não interferir no GC ou depender de seu comportamento é uma boa prática. Programadores mais acostumados a um controle fino do gerenciamento de memória podem se sentir desconfortáveis em deixar tudo na mão da VM. Mas a grande vantagem do gerenciamento automático de memória é justamente não mais nos preocuparmos com detalhes do GC e ter uma implementação eficiente à nossa disposição.

2.3 JIT Compiler: compilação em tempo de execução

Quando o Java foi lançado, a JVM era ordens de magnitude mais lenta que hoje, interpretando seu código que qualquer programa nativo em C. Hoje, o Java é bastante rápido, mas ainda carrega este estigma originado no passado.

As primeiras JVMs eram simplesmente interpretadores de bytecodes. Liam arquivos *.class* e traduziam cada bytecode na sequência de execução para as instruções de máquina equivalentes. A partir do Java 1.1, a Sun incluiu seu primeiro **JIT Compiler** , adquirido da Symantec, e, no Java 1.2, incluiu o HotSpot. A evolução dessas duas tecnologias ao longo das versões do Java é o que faz a JVM ser equiparável a aplicações nativas em C e C++.

Os grandes ganhos de performance do Java devem-se à otimização adaptativa, isto é, a capacidade de fazer otimizações de acordo com o uso do programa e as

condições do momento. Tanto Java quanto C possuem uma etapa de compilação estática, que gera um arquivo binário a ser executado, e, durante essa compilação, várias otimizações são realizadas. A diferença é que o binário nativo C gerado é executado diretamente na máquina, e o binário Java é ainda executado através da JVM, onde outras otimizações acontecem. No caso do C, toda otimização possível é feita estaticamente pelo compilador, enquanto que o Java consegue executar uma segunda etapa de otimizações dentro da VM no momento da compilação dos bytecodes para código nativo, na chamada compilação dinâmica. É isto o que faz o Java executar tão rapidamente.

Durante a execução do programa, a JVM a analisa e detecta os pontos mais importantes e executados da aplicação. Esses métodos são então compilados dinamicamente pelo JIT para código nativo otimizado, e toda execução subsequente é feita com ele, atingindo performance comparável a C. As VMs mais modernas conseguem até recompilar determinado método com otimização mais agressiva caso percebam que a performance melhorará. Elas também percebem que determinado código já não é mais importante, e descartam suas compilações, ou que determinada heurística aplicada não foi boa, e aplicam um novo tipo de otimização.

A evolução constante da JVM e do JIT faz com que o Java seja cada vez mais rápido. Mesmo um programa antigo melhora significativamente sua performance se o usuário simplesmente atualizar a versão da JVM. Em um programa nativo tradicional, para o usuário ter um ganho de performance, por exemplo, por causa de uma versão nova do compilador C, ele terá que esperar do fabricante uma versão nova recompilada. Mais do que isso, muitos programas comerciais em C exigem um mínimo de portabilidade, como, por exemplo, rodar desde o Windows XP até o Windows 7, ou suportar qualquer processador Intel e AMD. Para tanto, frequentemente a compilação do programa C é feita usando recursos comuns a todas as plataformas visadas. Ou seja, um denominador comum em relação a otimizações, instruções de máquina e chamadas do sistema. Já o Java, por não fazer a compilação para código nativo apenas estaticamente, consegue usar estratégias focadas no sistema operacional e no hardware usados no momento de sua execução, chegando a invocar instruções específicas do processador em uso.

Otimizações para clientes e servidores

Diferentes aplicações têm diferentes necessidades de performance e responsividade. Pensando nisso, a JVM oferece perfis diferentes que adaptam o funcionamento do JIT e outras características da JVM.

Aplicações voltadas ao usuário final, geralmente clientes Desktop, têm uma grande demanda por inicialização rápida, embora não precisem tanto de performance; em geral, o gargalo nessas aplicações é o input do usuário. Se habilitarmos a opção `-client` na JVM, ela perde menos tempo no startup para o usuário ver o resultado mais rapidamente, mas ao custo de ter um JIT fazendo otimizações menos agressivas a longo prazo. Um dos focos desta configuração é, por exemplo, não consumir muita memória, até deixando de fazer certas otimizações no JIT que acabariam aumentando o uso de memória, como *inline* de métodos.

Já a opção `-server` prevê cenários nos quais o tempo inicial não é um grande problema, mas existe uma preocupação maior com a performance a longo prazo. O startup demora mais algum tempo, e o JIT roda em uma de suas versões mais agressivas para o desempenho da aplicação.[38] O uso de memória é mais intenso, mas a performance de execução é maior a longo prazo. A VM server abusa, por exemplo, de recursos como *inline* de métodos, em que o código compilado pelo JIT já junta na memória o código dos métodos envolvidos para evitar *jumps* ao máximo.

Nas versões mais recentes da HotSpot, caso não passemos a opção explicitamente, a própria máquina virtual escolhe entre client e server, tamanho do heap, algoritmo de GC e outros pontos com base no tipo de hardware no qual o software está sendo executado – este recurso é chamado de VM *ergonomics*.[37] Há ainda opções experimentais, como a `-XX:+AgressiveOpts` que habilita mais otimizações.

Se desenhássemos um gráfico que mostrasse a relação entre performance e tempo para as VMs server e client, e ainda um programa C, teríamos algo como a Figura 2.3, a seguir:
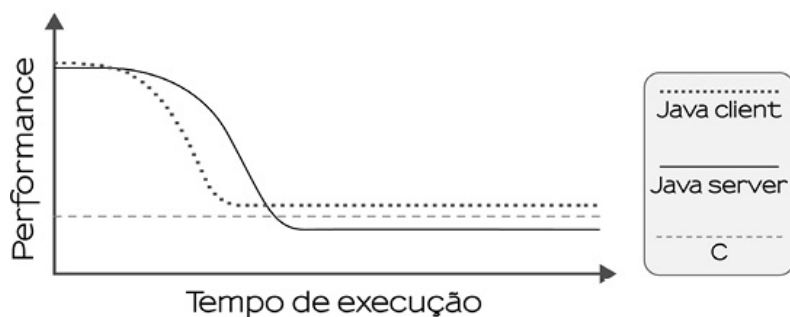
Figura 2.3: Ilustração da performance de uma aplicação ao longo do tempo.

Fácil de ser notado à primeira vista é o processo do startup demorado em Java. A

inicialização da JVM tem melhorado bastante a cada versão, mas ainda é um gargalo importante em aplicações curtas ou que tenham uma necessidade de início rápido. Essa demora se deve à quantidade de classes que a JVM precisa carregar no começo, e também ao tempo para esquentar o JIT. Na HotSpot, por exemplo, o programa começa executando tudo em modo interpretado e, só depois de identificar os pontos quentes, é que o JIT é acionado para determinado método. Enquanto isso, na JVM server, por padrão, o JIT faz a otimização após dez mil invocações. Já na JRockit, há um primeiro JIT compiler que roda desde o início realizando poucas otimizações e, conforme os pontos quentes surgem, um segundo JIT mais agressivo recompila os métodos escolhidos.

Dado este monitoramento que a JVM faz para decidir o momento certo de otimizar seu código pelo JIT, fica a pergunta: o que é melhor, escrever pequenos métodos reaproveitáveis entre si com muitas invocações, ou métodos gigantes que são invocados umas poucas vezes? Quebrar nosso código em pequenos métodos, que concentram uma única responsabilidade, é a melhor prática. E, por questões de performance, para o JIT, quanto mais reaproveitáveis forem seus métodos, mais invocações serão feitas e mais focadas serão as otimizações. *__Programe baseado em boas práticas, e deixe o JIT cuidar da execução otimizada__.

Outro ponto a se notar é a performance constante na aplicação C. No início, ela é extremamente mais rápida, mas toda otimização possível em um programa C foi realizada em compilação. Uma aplicação em Java pode ter sua performance variável ao longo da execução, de acordo com a atividade do JIT. Em uma aplicação que não exija tanto I/O e threads, a JVM é madura o suficiente para ser equiparável com C na maioria das situações.

É possível visualizar o comportamento do JIT usando algumas opções avançadas da JVM. Na HotSpot, a opção `-XX:+PrintCompilation` habilita um modo onde todo método compilado pelo JIT é impresso no console. Ao testar em aplicações bem simples, com poucas invocações, é possível notar que o JIT raramente é acionado. Em outras com mais invocações de métodos (um bom teste é usar algum algoritmo recursivo), nota-se que os métodos mais quentes são compilados pelo JIT.

Mas como saber os métodos a serem compilados? Na HotSpot, ao usar a VM client, o padrão é aguardar 1.500 invocações de um método até compilá-lo. Já na VM server, são aguardadas 10.000 invocações antes da compilação. É, aliás, o que nosso gráfico anterior indicava, que a VM server demora um pouco mais para esquentar, mas tem uma performance melhor a longo prazo.

Na HotSpot, é possível controlar o número de invocações a serem aguardadas

antes das compilações usando a opção `-XX:CompileThreshold=<NUM>` passando o número desejado. Muitas pessoas perguntam-se por que o JIT não compila todos os métodos, rodando tudo de forma nativa e otimizada. É fácil testar este caso, passando 1 na opção anterior, e logo você notará uma perda de performance em muitas aplicações pequenas ou com pouca lógica repetida. Neste caso, o custo de compilação e otimização é muito alto. Em outros, principalmente em aplicações mais longas, é possível até ter algum ganho, mas logo percebe-se que os números calculados por padrão pela JVM acabam sendo valores bem razoáveis para a maioria das aplicações. Outro teste interessante é usar a opção `-Xint`, que desabilita completamente o JIT e faz a VM rodar apenas em modo interpretado. Em aplicações um pouco mais longas, logo se percebe uma imensa perda de performance.

De modo geral, o que se percebe é que não ter um JIT é uma péssima ideia, quase tão ruim quanto compilar todos os métodos existentes. Na prática, o melhor acaba sendo o comportamento padrão do JIT de compilar apenas os pontos quentes. Nossa aplicação, porém, deve, sim, estar preparada para usufruir o melhor do JIT. Em geral, isso significa as boas práticas de design com vários pequenos métodos sendo reaproveitados por todo o código.

Pode ser necessário monitorar a execução de uma aplicação, como quando há a necessidade de encontrar gargalos específicos de performance e vazamentos de memória. A Oracle disponibiliza diversas ferramentas para facilitar o *profiling*, como jmap, jstat e jconsole.[34] Plugins para IDEs também existem em abundância, como o TPTP do Eclipse. Dentre as ferramentas pagas, destacam-se o TestKit e o JProfiler. Todos eles permitem a um desenvolvedor extrair informações de uma aplicação rodando na plataforma Java para entender melhor como ela está funcionando em produção, até mesmo aquelas escritas em outras linguagens, mas interpretadas na VM.

2.4 Carregamento de classes e classloader hell

É frequente diversas aplicações Web serem implantadas em um mesmo Servlet Container, em uma mesma máquina virtual, e também que muitas delas utilizem bibliotecas e frameworks em comum. Imagine duas aplicações que utilizam a mesma versão do Hibernate: onde devemos colocar os JARs?

Muitos sugeriram colocá-los em algum diretório common/lib do Servlet Container; outros disseram para jogar diretamente na variável de ambiente `CLASSPATH`. Esses eram conselhos frequentes, em especial quando era raro ter mais de uma apli-

cação Java implantada em um mesmo servidor. Para entender os graves problemas que podem surgir nesta situação, é preciso primeiro compreender como é o funcionamento do carregador de classes da JVM.

Classloader é o objeto responsável pelo carregamento de classes Java para a memória a partir de um array de bytes que contém seus bytecodes.[114] É representado pela classe abstrata `java.lang.ClassLoader`, e existem diversas implementações concretas, como a `java.net.URLClassLoader`, responsável por carregar classes buscando em determinadas URLs.

Classloaders são um ponto-chave da plataforma Java. Com eles, podemos carregar diferentes classes com exatamente o mesmo nome e mantê-las de forma distinta, em diferentes espaços de nomes (*namespaces*). Basta que usemos classloaders diferentes. Uma classe é unicamente identificada dentro da JVM por seu nome completo (incluindo o nome do pacote, o chamado *fully qualified name*) mais o classloader que a carregou.

É desta forma que servidores de aplicação conseguem manter separadas versões diferentes de uma mesma biblioteca; podemos ter o Hibernate 3.6 em uma aplicação Web e o Hibernate 3.3 em outra. Para isso, dois classloaders diferentes são necessários para carregar as classes das duas diferentes aplicações. Como um classloader carrega uma classe uma única vez, se neste caso fosse utilizado o mesmo classloader, haveria conflito de classes com o mesmo nome (por exemplo, a `org.hibernate.Session` das duas versões de Hibernate diferentes), prevalecendo apenas a primeira que fosse encontrada.

Porém, o mecanismo de carregamento de classes não é tão simples. Por uma medida de segurança, os classloaders são criados e consultados de maneira hierárquica (Figura [ref-label classloaders2-4).
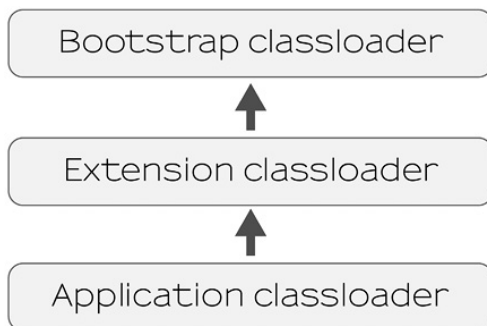
Figura 2.4: Hierarquia de classloaders.

A figura mostra os três classloaders que a JVM sempre utilizará por padrão. Antes de um classloader iniciar um carregamento, ele pergunta ao seu pai (*parent classloader*) se não consegue carregar (ou já carregou) a classe em questão. Se o *parent classloader* conseguir, ele será responsável por esse carregamento. Esse funcionamento hierárquico é o que garante segurança à plataforma Java. É assim que classes importantes, como as do `java.lang`, serão sempre lidas pelo *bootstrap classloader*, caso contrário, poderíamos, por exemplo, adicionar uma classe `java.net.Socket` à nossa aplicação e, quando esta fosse implantada no servidor, todas as outras classes dessa aplicação que fizessem uso da Socket estariam usando agora um provável cavalo de troia.

Logo, é vital um certo conhecimento desses classloaders fundamentais.

- **Bootstrap classloader**: Carrega classes do rt.jar, e é o único que
- realmente não tem *parent* algum; é o pai de todos os outros, evitando que qualquer classloader, tente modificar as classes do rt.jar (pacote `java.lang`, por exemplo).
- **Extension classloader** Carrega classes de JARs dentro do diretório na
- propriedade `java.ext.dirs`, que, por padrão, tem o valor `$JAVA_HOME/lib/ext`. Na JVM da Sun, é representado pela classe interna `sun.misc.Launcher$ExtClassLoader`. Tem como pai o *Bootstrap classloader*.
- **Application classloader ou System classloader**: Carrega tudo definido no

- `CLASSPATH`, tanto da variável de ambiente, quanto da linha de comando (passado como argumento por `-cp`). Representado pela classe interna `sun.misc.Launcher$AppClassLoader`, tem como pai o *Extension classloader*.

O Bootstrap classloader é o que carrega todas as classes da biblioteca padrão (rt.jar inclusive). Depois dele, a aplicação pode usar outros classloaders, já que a maioria das classes está fora desse JAR.

As aplicações podem criar novos classloaders e incrementar essa hierarquia (Figura 2.5). Os containers costumam criar muitos outros classloaders além dos padrões da máquina virtual. Em geral, criam um **Container classloader**, responsável pelos JARs do container e de uma certa pasta, como uma *common/lib*. Além deste, o container precisa de um classloader específico para cada aplicação (contexto) implantada. Esse **WebApplication** classloader é responsável pelo carregamento das classes do `WEB-INF/classes` e do `WEB-INF/lib` em aplicações Web, por exemplo.

Conhecendo essa hierarquia, podemos perceber o problema quando desenvolvedores desavisados jogam JARs em diretórios compartilhados por todas as aplicações de um determinado servidor, como o caso do diretório `common/lib` do Apache Tomcat em versões muito antigas, ou utilizando a variável de ambiente `CLASSPATH`.

Considerando que temos duas aplicações, APP-nova e APP-antiga, que usam a mesma versão do Hibernate 3.3, elas funcionarão normalmente mesmo com os jars do Hibernate referenciados na variável `CLASSPATH`. No momento que a APP-nova precisar utilizar a versão 3.6, como proceder? Jogar a versão mais nova no `WEB-INF/lib` da APP-nova não resolve o problema, pois a hierarquia de classloaders faria com que a `org.hibernate.Session` e todas as outras classes do Hibernate sejam carregadas dos jars definidos na `CLASSPATH`, possivelmente gerando erros inesperados em tempo de execução (Figura 2.5).
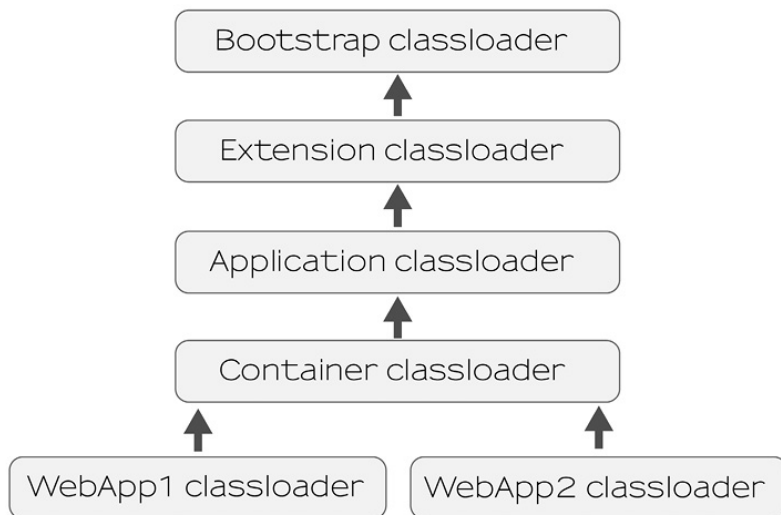
Figura 2.5: Hierarquia usual de classloaders no container.

Esse problema é muito mais grave do que aparenta; o tipo de erro que pode surgir é de difícil interpretação. O menor dos problemas seria ter o comportamento da versão antiga do Hibernate. Mais grave é quando há métodos novos no JAR mais recente; a interface Session do Hibernate antigo seria carregada, e quando sua aplicação APP-nova invocar um método que só existe na versão nova, `NoSuchMethodError` será lançado! Isso causa uma enorme dor de cabeça, pois o desenvolvedor fica confuso ao ver que o código compilou perfeitamente, mas durante a execução a JVM indica que aquele método não existe.

`NoSuchMethodError` é um **forte indicador de que o código foi compilado esperando uma versão diferente de uma biblioteca que a encontrada em tempo de execução**, provavelmente por causa de os jars estarem espalhados e compartilhados.

O comportamento da aplicação pode ser mais errático; imagine que a aplicação APP-nova utiliza diretamente uma classe do Hibernate que só existe na versão mais nova, como a `TypeResolver`. O classloader que verifica a variável `CLASSPATH` não a encontrará, e então ela será corretamente carregada do `WEB-INF/lib` pelo classloader específico do contexto. Porém, a `TypeResolver` do Hibernate 3.6 referencia outras classes básicas do Hibernate, e estas serão carregadas da versão antiga, o Hibernate encontrado na `CLASSPATH`, fazendo com que o Hibernate seja carregado parte de uma versão, parte de outra, resultando em efeitos colaterais inesperados e desastrosos.

Essa confusão é similar ao DLL Hell, frequentemente chamado de **Classloader hell** na plataforma Java.

Em algumas configurações de classloaders diferentes, que carregam classes de diretórios em comum, pode aparecer um `ClassCastException` curioso e de difícil discernimento. Se uma referência a um objeto do tipo Produto for passada como argumento para um método que recebe um objeto também deste tipo, mas esta classe tiver sido carregada por um classloader diferente, a exceção será lançada. O desenvolvedor ficará confuso ao ver uma `ClassCastException` em uma invocação de método em que nem mesmo há um casting. Isso acontece porque, como vimos, em tempo de execução a identidade de uma classe não é apenas seu *fully qualified name*, mas também o classloader que a carregou. Assim, uma classe com mesmo nome, mas carregada por classloaders diferentes, é outra. Isto é chamado de **runtime identity** e muito importante para permitir que containers tenham mais de uma versão da mesma classe na memória (como o exemplo das `Sessions` de diferentes versões do Hibernate).[48]

Muitos containers, incentivados pela própria especificação de Servlets, aplicam um modelo de classloaders invertidos para minimizar os problemas. Ao invés de deixar o WebApp classloader filho do Container classloader, ele é feito filho direto do Bootstrap classloader mas com uma referência simples para o Container classloader. A arquitetura tradicional dificulta que uma aplicação sobrescreva algum componente do Container classloader, mas esse modelo invertido (padrão no Tomcat, por exemplo) permite esse cenário. A ordem de resolução das classes passa a ser: primeiro o Bootstrap, depois as classes da aplicação e depois as compartilhadas do Container.

Criando seu ClassLoader

Podemos instanciar a `URLClassLoader` para carregar classes a partir de um conjunto dado de URLs. Considere que a classe `DAO` está dentro do diretório bin:

```
URL[] bin = {new URL("file:bin/")};
ClassLoader loader = new URLClassLoader(bin, null);
Class<?> clazz = loader.loadClass("br.com.arquiteturajava.DAO");
```

Repare que passamos null como segundo argumento do construtor de `URLClassLoader`. Isto indica que seu parent classloader será o bootstrap diretamente.

Se também carregarmos a classe através do `Class.forName` e compararmos a referência das duas classes:

```
URL[] bin = {new URL("file:bin/")};
ClassLoader loader = new URLClassLoader(bin, null);

Class<?> clazz = loader.loadClass("br.com.arquiteturajava.DAO");
Class<?> outraClazz = Class.forName("br.com.arquiteturajava.DAO");

System.out.println(clazz.getClassLoader());
System.out.println(outraClazz.getClassLoader());
System.out.println(clazz == outraClazz);
```

O `Class.forName()` carrega a classe usando o classloader que foi responsável pela classe do código executado; no caso, o Application classloader se estivermos no método main. E o resultado com a JVM da Oracle/Sun é:

```
java.net.URLClassLoader@19821f
sun.misc.Launcher$AppClassLoader@11b86e7
false
```

Executando este mesmo código para uma classe da biblioteca padrão, digamos, `java.net.Socket` ou `java.lang.String`, teremos um resultado diferente. Elas serão carregadas em ambos os casos pelo *bootstrap* classloader (representado pelo `null`), pois não há como evitar que esse classloader seja consultado em razão da questão de segurança já discutida.

Ao remover o `null` do construtor, o `URLClassLoader` terá como parent o classloader que carregou o código sendo executado (o *Application classloader*), fazendo com que o carregamento da classe seja delegado para este, antes de ser tentado pelo nosso `URLClassLoader`. Caso o diretório bin esteja no classpath, o *Application classloader* consegue carregar a classe DAO e o nosso recém-criado classloader não será o responsável pelo carregamento:

```
URL[] bin = {new URL("file:bin/")};
ClassLoader loader = new URLClassLoader(bin);
```

Desta vez, o resultado é:

```
sun.misc.Launcher$AppClassLoader@11b86e7
sun.misc.Launcher$AppClassLoader@11b86e7
true
```

O *parent* classloader será sempre consultado antes de o classloader de hierarquia “mais baixa” tentar carregar uma determinada classe. Lembre-se de que este é o motivo pelo qual jogar os JARs na variável de ambiente `CLASSPATH` pode acabar escondendo versões diferentes da mesma classe que estejam ao alcance de classloaders mais “baixos” na hierarquia.

Endorsed JARs

Esse funcionamento hierárquico causa problemas também com a biblioteca padrão; a cada versão nova do Java, algumas APIs, antes externas e opcionais, são incorporadas. Este foi o caso, por exemplo, das APIS de parseamento de XML do Apache (Xerces e Xalan). Caso elas fossem incluídas dentro da biblioteca padrão, qualquer outro projeto que necessitasse delas em uma versão diferente (tanto mais atual quanto mais antiga) teria sempre a versão do rt.jar

Para contornar o problema, a Sun colocou as classes que seriam do pacote org.apache para dentro de `com.sun.org.apache`.[136] Sem dúvida uma maneira deselegante, mas que evitou o problema de versionamento.

Este problema tornou-se ainda mais frequente. No Java 6, a API de mapeamento de XML, a JAXB 2.0, entrou para a biblioteca padrão. Quem necessita usar a versão 1.0, ou ainda uma futura versão 3.0, terá problemas; a versão 2.0 sempre terá prioridade no carregamento. O JBoss 4.2 necessita da API do JAXB 1.0 para sua implementação de Web Services, e sofre então com essa inclusão do Java 6.[180] Aliás, o JBoss possui uma longa e interessante história de manipulação dos classloaders, sendo o primeiro EJB container a oferecer *hot deployment*, e para isto teve de contornar uma série de restrições e até mesmo bugs da JVM.[24]

Quando o interpretador de Javascript Rhino entrou junto com a Scripting API no Java 6 na JVM da Sun, passamos a ter o mesmo problema quando é necessário utilizar suas versões diferentes.[112]

Nesses dois casos, para contornar o problema, utilizamos o recurso de endorsed JARs. Através da linha de comando (`-Djava.endorsed.dirs=`), você especifica diretórios que devem ter prioridade na procura de classes antes que o diretório ext do Java SE seja consultado. É uma forma de o administrador do sistema dizer que confia em (endossa) determinados JARs. Alguns servidores de aplicação possuem um diretório especial onde você pode jogar os JARs a serem endossados. Vale notar o cuidado que deve ser tomado ao confiar em um JAR, colocando-o no topo da hierarquia dos classloaders.

OutOfMemoryError no redeploy de aplicações

Vimos que algumas JVMs armazenam as classes carregadas na memória no espaço chamado **PermGen**. É comum aparecerem problemas com `OutOfMemoryError`, acusando que este espaço se esgotou, dado um número muito grande de classes carregadas. Em particular, isso ocorre facilmente depois de alguns hot deploys em um container.

Toda classe carregada tem uma referência para o seu classloader, assim como todo classloader referencia todas as classes carregadas por ele. Isso para que possa devolver sempre a mesma classe no caso de outra invocação subsequente para o mesmo *full qualified name*, agindo como uma factory que cacheia suas instanciações. Esse relacionamento bidirecional entre `Class` e `ClassLoader` faz com que os objetos `Class` só sejam coletados pelo garbage collector junto com o classloader inteiro e todas as outras classes associadas. Logo, a única maneira de um objeto `Class` ser coletado é se todas as referências para todas as classes do seu classloader forem liberadas também.

Ao realizar o hot deploy de uma aplicação, o próprio container libera as referências das classes antigas, possibilitando a coleta do classloader do contexto. Isso deveria ser suficiente para que o contexto fosse inteiramente liberado da memória, mas, na prática, outro classloader acima daquele da WebApplication segura referências para classes da aplicação. E há várias bibliotecas que assim se comportam, como o próprio DriverManager do JDBC (*Java Database Connectivity*). Este guarda referências para as classes dos drivers registrados, mas elas são carregadas pelo bootstrap, e o driver, em geral, está dentro da aplicação, no `WEB-INF/lib`. O carregamento de um único driver JDBC é capaz de segurar na memória o contexto inteiro de uma aplicação que não é mais necessária.

Existem vários outros exemplos de bibliotecas e classes que causam memory leaks parecidos[5] [194]. No caso do JDBC, a solução é fazer um *context listener* que invoca o método `deregisterDriver` no `DriverManager`, mas há situações não tão simples de se contornar, que envolvem versões específicas de bibliotecas com seus próprios tipos de leaks de referências a classes que já não são mais utilizadas. Na prática, é quase impossível uma aplicação Java conseguir evitar esses leaks de classloaders, por isso em algum momento surgem os `OutOfMemoryError` frequentes nos hot deploys, sendo mais um motivo para evitá-los em produção.

Capítulo 3

Tópicos de Orientação a Objetos

Um bom design de software visa a uma arquitetura flexível que permita futuras alterações, facilite a produção de código organizado e legível, maximizando seu reaproveitamento. Todo o paradigma da orientação a objetos, seus princípios e boas práticas procuram trazer esses benefícios para o design.

Ao pensar no sistema como um todo, outras questões de mais alto nível surgem, em especial aquelas que tratam da forma como os objetos se relacionam e sua organização dentro e entre sistemas. Ao relacionar dois objetos distintos, deve-se levar em conta as boas práticas que serão discutidas nesse capítulo, como a diminuição do acoplamento entre objetos.

3.1 Programe voltado à interface, não à implementação

Ao trabalhar com coleções, escolher a implementação certa para cada caso é uma tarefa difícil. Cada uma delas, como `ArrayList`, `LinkedList` ou `HashSet`, é melhor

para resolver determinas categorias de problemas. Pode ser muito arriscado escrever todo o código da aplicação dependente de uma decisão antecipada. Apesar disso, grande parte dos desenvolvedores opta por sempre utilizar `ArrayList` desde o início sem critério algum.

Considere um `DAO` de funcionários que pode listar o nome de todos que trabalham em determinado turno, devolvendo um `ArrayList` com os nomes:

```
public class FuncionarioDao {
    public ArrayList<String> buscaPorTurno(Turno turno) { ... }
}
```

E um código que precisa saber se determinado funcionário esteve presente, efetuando uma busca simples na lista devolvida:

```
FuncionarioDao dao = new FuncionarioDao();
ArrayList<String> nomes = dao.buscaPorTurno(Turno.NOITE);

boolean presente = nomes.contains("Anton Tcheckov");
```

Mas a busca com `contains` em um `ArrayList` é, em termos computacionais, bastante custosa. Poderíamos utilizar outras alternativas de coleção, trocando o retorno de `ArrayList` para `HashSet`, por exemplo, pois sua operação `contains` é muito mais eficiente computacionalmente, usando internamente uma tabela de espalhamento (*hash table*). Para poucos funcionários, a diferença é imperceptível, porém, à medida que a lista aumenta, a diferença de desempenho entre um `ArrayList` e um `HashSet` se torna mais clara, e até mesmo um gargalo de performance.

O problema em realizar uma mudança como esta, de implementação, é que todo código que usava o retorno do método como `ArrayList` quebra, mesmo que só usássemos métodos que também existem definidos em `HashSet`. Seria preciso alterar todos os lugares que dependem de alguma forma desse método. Além de trabalhoso, tarefas do tipo `search/replace` são um forte sinal de código ruim.

Há esse acoplamento sintático com a assinatura do método, que conseguimos resolver olhando os erros do compilador. Mas sempre há também informações semânticas implícitas na utilização desse método, e que não são expostos através da assinatura. Um exemplo de acoplamento semântico está em depender da informação de que uma `List` permite dados duplicados, enquanto um `Set` garante unicidade dos elementos. Como problemas no acoplamento sintático são encontrados em tempo de compilação, os semânticos somente o são em execução, daí um motivo da importância de testes que garantam o comportamento esperado.

O grande erro do método `buscaPorTurno` da classe `FuncionarioDao` foi atrelar todos os usuários do método a uma implementação específica de `Collection`. Desta forma, alterar a implementação torna-se sempre muito mais custoso, caracterizando o **alto acoplamento** que tanto se procura evitar.

Para minimizar esse problema, é possível usar um tipo de retorno de método mais genérico, que contemple diversas implementações possíveis, fazendo com que os usuários do método não dependam em nada de uma implementação específica. A interface `Collection` é uma boa candidata:

```
public class FuncionarioDao {
    public Collection<String> buscaPorTurno(Turno turno) { ... }
}
```

Com o método desta forma, podemos trocar a implementação retornada sem receio de quebrar nenhum código que esteja invocando `buscaPorTurno`, já que ninguém depende de uma implementação específica. Usar *interfaces* Java é um grande benefício nestes casos, pois ajuda a garantir que nenhum código dependa de uma implementação específica, pois interfaces não carregam nenhum detalhe de implementação.

Repare que é possível optar ainda por outras interfaces, como `List` (mais específica) e `Iterable` (menos específica). A escolha da interface ideal vai depender do que você quer permitir que o código invocador possa utilizar e realizar na referência retornada. Quanto menos específica, menor o acoplamento e mais possibilidades de diferentes implementações. Em contrapartida, o código cliente tem uma gama menor de métodos que podem ser invocados.

Algo similar também ocorre para receber argumentos. Considere um método que grava diversos funcionários em lote no nosso `DAO`:

```
public class FuncionarioDao {
    public void gravaEmLote(ArrayList<Funcionario> funcionarios) { ... }
}
```

Receber precisamente `ArrayList` como argumento tem pouca utilidade; raramente necessitamos que uma coleção seja tão específica assim. Receber aqui um `List` provavelmente baste para o nosso método, e permite que o código invocador passe outras coleções como argumento. Podemos ir além e receber `Collection` ou, ainda, `Iterable`, caso nossa necessidade seja apenas percorrer os elementos. A escolha de `Iterable`, neste caso, permitiria o maior desacoplamento possível, mas limitaria o uso dentro do método; não seria possível, por exemplo, acessar a quantidade

de elementos que essa coleção possui, nem os elementos de maneira aleatória através de um índice. Devemos procurar um balanço entre o desacoplamento e a necessidade do nosso código. Esta é a ideia do *Princípio de Segregação de Interfaces* : **clientes não devem ser forçados a depender de interfaces que não usam**.[122]

O desenvolvedor deve ter em mente que acoplar uma classe, que possui menos chances de alterações em sua estrutura, com outra menos *estável* pode ser perigoso. [125] Considere a interface `List`, que possui muitas razões para não mudar, afinal, ela é implementada por várias outras classes; se sofresse alterações, todas as classes que a implementam teriam que ser alteradas também. Consideramos, então, que ela é altamente estável, o que significa que ela raramente obrigará uma mudança nas classes que a utilizam (Figura 3.1).
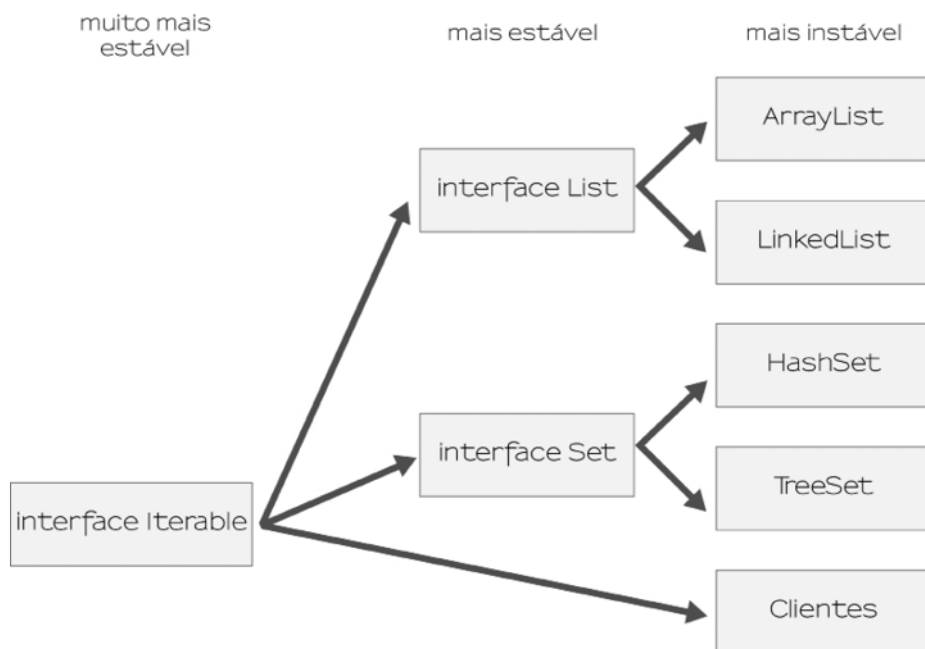
Figura 3.1: Interfaces são mais estáveis por garantirem menores mudanças com quebra de compatibilidade.

Uma implementação de lista, `MeuProprioArrayList`, feita pelo desenvolvedor é provavelmente mais *instável* que a interface `List`, já que as forças que a impedem de mudar são fracas (não há outras classes utilizando essa implementação). Ou seja,

uma classe acoplada a essa implementação de lista eventualmente pode ser obrigada a mudar por causa de alguma alteração em `MeuProprioArrayList`.

Os frameworks e bibliotecas consagrados sempre tiram proveito do uso de interfaces, desacoplando-se o máximo possível de implementações. Tanto a `Session` do Hibernate quanto a `EntityManager` da JPA devolvem `List` nos métodos que envolvem listas de resultados. Ao analisar a fundo as implementações atuais de `Session` e `EntityManager` do Hibernate, elas não retornam nem `ArrayList`, nem `LinkedList`, nem nenhuma coleção do `java.util`, e, sim, implementações de listas persistentes de pacotes do próprio Hibernate.

Isto é possível, novamente, pelo desacoplamento provido pelo uso das interfaces. Além disso, o retorno das consultas com JPA e Hibernate são `List`, para deixar claro ao usuário que a ordem é importante. Manter a ordem de inserção e permitir acesso aleatório são características do contrato de `List` e são importantes para o resultado de consultas, pois podem definir uma ordenação (`order by`), uma ótima justificativa para optar por uma interface mais específica, e não usar `Iterable` ou `Collection`.

Nas principais APIs do Java, é fundamental programar voltado à interface. Ao usar `java.io`, evitamos ao máximo nos referenciar a `FileInputStream`, `SocketInputStream`, entre outras. O código a seguir aceita apenas arquivos como streams:

```
public class ImportadoraDeDados {
    public void carrega(FileInputStream stream) { ... }
}
```

Desta forma, não é possível passar qualquer tipo de `InputStream` para a `ImportadoraDeDados`. Adotar esta limitação depende do código dentro do método `carrega`. Ao utilizar algum método específico de `FileInputStream` que não esteja definido em `InputStream`, não há o que fazer para desacoplar o código. Caso contrário, esse método poderia, e deveria, receber uma referência a `InputStream`, ficando mais flexível e podendo receber os mais diferentes tipos de streams, como argumento, que provavelmente não foram previamente imaginados. **Utilize sempre o tipo menos específico possível**.

Repare que, muitas vezes, classes abstratas trabalham como interfaces, no sentido conceitual de orientação a objetos. [206] Classes abstratas possuem a vantagem de se poder adicionar-lhes um novo método não abstrato, sem quebrar o código já existente. Já com o uso de *interfaces* (aqui, pensando na palavra-chave do Java), a adição de qualquer método acarretará a quebra das classes que a implementam. Como

interfaces nunca definem nenhuma implementação, a vantagem é que o código está sempre desacoplado de qualquer outro que as utilize. Isso muda com os polêmicos *extension methods* do Java 8, permitindo escrever uma implementação padrão nas interfaces, possibilitando suas evoluções, ao mesmo tempo que minimiza a quebra de compatibilidade.

Os métodos `load(InputStream)`, da classe `Properties`, e `fromXML(InputStream)`, do XStream, são ótimos exemplos de código que não dependem de implementação. Podem receber arquivos dos mais diferentes streams: rede (`SocketInputStream`), upload HTTP (`ServletInputStream`), arquivos (`FileInputStream`), de dentro de JARs (`JarInputStream`) e arrays de byte genéricos (`ByteArrayInputStream`).

A *Java Database Connectivity* (JDBC) é outra API firmemente fundada no uso de interfaces. O pacote `java.sql` possui pouquíssimas classes concretas. Sempre que encontramos um código trabalhando com conexões de banco de dados, vemos referências à interface `Connection`, e nunca diretamente a `MySQLConnection`, `OracleConnection`, `PostGreSQLConnection`, ou qualquer outra implementação de um driver específico, apesar de esta possibilidade existir.

Referindo-se sempre a `Connection`, deixamos a escolha da implementação centralizada em um ou poucos locais. Desta forma, fica muito fácil trocar a implementação sem modificar todo o restante do código. Isso ocorre graças ao desacoplamento provido pelo uso de interfaces.

No caso do JDBC, essa escolha por uma implementação está centralizada na classe concreta `DriverManager`, que aqui age como uma factory. Ela decide por instanciar uma implementação específica de `Connection` de acordo com os parâmetros passados como argumentos ao método `getConnection` e conforme os possíveis drivers previamente carregados.

```
Connection connection =
    DriverManager.getConnection("jdbc:mysql://192.168.0.33/banco");
```

Pode ser fácil enxergar as vantagens do uso das interfaces, mas é bem mais difícil começar a utilizá-las extensivamente no seu próprio domínio. O uso exagerado de reflection para invocar métodos dependendo de algumas condições pode ser muitas vezes substituído por interfaces. Assim, a decisão de qual método invocar é deixada para a invocação virtual de método que o polimorfismo promove, diminuindo bastante a complexidade e aumentando a manutenibilidade, além de algum ganho de performance. [17]

Isto é ainda mais gritante com o uso da instrução `switch`, ou mesmo em um excessivo número de `ifs` encadeados; essa abordagem pode acoplar totalmente seu modelo, tornando necessárias mudanças frequentes nele toda vez que uma nova entidade for adicionada ao domínio.[66]

Programe voltado à interface, não à implementação é outro dos príncipios de orientação a objetos do livro *Design Patterns* ([206] [79]), abordado por meio de outros exemplos no seminal *Dependency Inversion Principle*, de Bob Martin. [121]

3.2 Componha comportamentos

Um código com poucas possibilidades de fluxos lógicos (*branches* de execução), ou seja, poucos caminhos de execução, é mais fácil de entender e manter. O exemplo a seguir mostra um processo de pagamento:

```
public void processa(Pagamento aPagar) {
    if (aPagar.isServico() && aPagar.getValor() > 300) {
        impostos.retem(aPagar.getValor() * TAXA_A_RETER);
    } else if(aPagar.isProduto()) {
        estoque.diminui(aPagar.getItem());
    }

    conta.executa(aPagar);

    if (aPagar.desejaReceberConfirmacao()) {
        emails.enviaConfirmacao(aPagar);
    }
}
```

Apesar de simples, existem seis possibilidades diferentes de execução do método, além de ele misturar diversos comportamentos que não possuem relação, isto é, responsabilidades diferentes para uma única classe: impostos, estoques, transferências e e-mails.

Tal comportamento pode ser composto por diversas partes menores e, para tanto, refatorações pequenas podem ser executadas. A mais simples seria a extração de quatro métodos, uma solução que simplifica o código atual, mas não aumenta sua coesão.

É possível aumentar a coesão da classe extraindo tais métodos em quatro classes distintas, como `RetemImposto`, `ProcessaEstoque`, `Transferencia` e `Confirmacao`. Por fim, as condições podem ser tratadas através de uma cadeia de responsabilidade

(*chain of responsibility*) ou notificações (*observers*), usando uma interface comum a todos esses processos que devem ser executados. Este design, que favorece a composição em vez de um grande método, isola preocupações distintas para partes separadas do código.

Uma interface `Processo` simples é capaz de modelar os diferentes tipos de processos que podem ser aplicados a um `Pagamento`, como a `RetemImposto`:

```
public interface Processo {
    void lidaCom(Pagamento pagamento);
}

public class RetemImposto implements Processo {

    private static final double TAXA_A_RETER = 0.10;

    public void lidaCom(Pagamento aPagar) {
        if (aPagar.isServico() && aPagar.getValor() > 300) {
            impostos.retem(aPagar.getValor() * TAXA_A_RETER);
        }
    }
}
```

Outras implementações possíveis de passos do processo seriam as classes `ProcessaEstoque`, `Transferencia` e `Confirmacao`. A cadeia de responsabilidade pode ser representada por uma outra classe `Processos`, que aplica diferentes processos a um mesmo pagamento de maneira encadeada:

```
public class Processos {

    private final List<Processo> processos;

    public Processo(List<Processo> processos) {
        this.processos = processos;
    }

    public void processa(Pagamento aPagar) {
        for (Processo passo : processos) {
            passo.lidaCom(aPagar);
        }
    }
}
```

Pode-se evoluir ainda mais o exemplo acima; quando a execução de cada passo de um comportamento é condicional e de acordo com as mais variadas regras, é comum adicionar um método extra de verificação, que indica se o processamento deve ou não ser feito naquele caso:

```
public interface Processo {
    void lidaCom(Pagamento pagamento);
    boolean deveProcessar(Pagamento pagamento);
}

public class RetemImposto implements Processo {
    private static final double TAXA_A_RETER = 0.10;

    public void lidaCom(Pagamento aPagar) {
        impostos.retem(aPagar.getValor() * TAXA_A_RETER);
    }

    public boolean deveProcessar(Pagamento aPagar) {
        return aPagar.isServico() && aPagar.getValor() > 300;
    }
}
```

Padrões de design parecidos com o anterior são encontrados nos conversores de diversos frameworks, renderização de telas, execução de processos complexos, filtros da servlet API etc. (Figura 3.2).
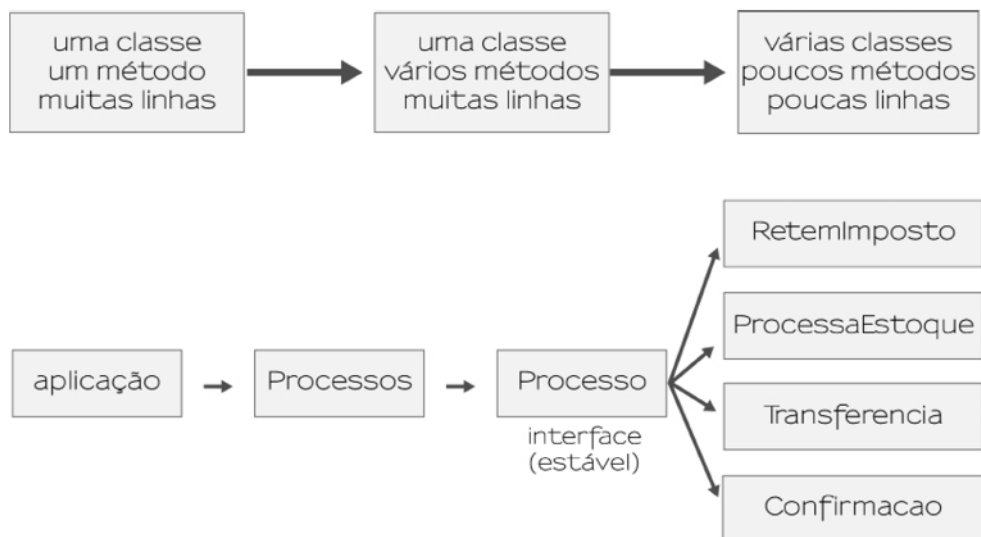
Figura 3.2: Compondo comportamento através de diferentes componentes de uma aplicação. À medida que foi necessário, o código era refatorado.

Após a quebra de um comportamento em diversas partes, é necessário juntar esses comportamentos novamente. Composição (*composition*) é esta união através da utilização de alguma prática de design, como no caso visto anteriormente, no qual o polimorfismo permite trabalhar de maneira uniforme com partes que executam tarefas distintas.

Um método que apresente um corpo com um número razoável de *branches* (como `ifs` e `switches`), ou que não permita a compreensão imediata de seu comportamento, pode ser refatorado para facilitar sua manutenção.

3.3 Evite herança, favoreça composição

O uso ou não de herança é um antigo debate que aparece inúmeras vezes na literatura, antes mesmo da existência do Java, ([79], [205]) e em recentes artigos com linguagens mais modernas. [206], [17], [91] Alguns críticos chegam a afirmar que ela nunca deveria ser utilizada. [192]

O maior problema com a herança é que acoplamos a implementação da classe mãe muito precocemente, criando a necessidade de a classe filha conhecer muito

bem o código interno da classe mãe, o que é visto como uma quebra de encapsulamento.

Podemos enxergar este problema em um exemplo encontrado no pacote `java.util`, com `Properties` e `Hashtable`. A classe `Properties` herda de `Hashtable` e, portanto, instâncias de `Properties` possuem disponíveis métodos de `Hashtable`. Mas, muitas vezes, esses métodos não fazem sentido, como no caso da `Properties` que lida apenas com `Strings`. A quebra aparece quando temos uma instância de `Properties` e invocamos o método `put(Object, Object)`, em vez de `setProperty(String, String)`, podendo passar um `Object` qualquer como argumento. Nesse caso, ao invocar posteriormente o método `getProperty(String)`, recebemos um inesperado `null`. Não há como evitar que alguém invoque o método que trabalha com `Object`, a não ser documentando a respeito, como ocorre no `javadoc` dessa classe. Ter de garantir condições apenas através do `javadoc` é um contrato muito fraco, em especial dizendo que tal método não deve ser invocado com determinados tipos de objetos, algo que, em Java, poderia ser feito através da tipagem estática.

```
public class TesteProperties {
    public static void main(String[] args) {
        Properties properties = new Properties();
        // não deveria ser possível invocar:
        properties.put("data_de_inicio" , Calendar.getInstance());

        Processador p = new Processador();
        p.processa(properties);
    }
}

public class Processador {
    public void processa(Properties properties) {
        String valor = properties.getProperty("data_de_inicio");
        // o Calendar estará na String?
    }
}
```

A relação *é-um* da herança nem sempre é tão fácil de ser aplicada a objetos. Apesar de um quadrado ser um retângulo, um carro ser um veículo e um dicionário de propriedades ser uma tabela de espalhamento, não necessariamente um objeto do tipo dicionário deve ser encarado como um objeto de tabela de espalhamento caso

isto não respeite o *princípio de substituição de Liskov*. [123] [91] Por este princípio, deveria ser possível receber um `Properties` como um `Map` sem nenhum problema, mas vimos que isso não ocorre no código descrito antes. Ainda no pacote `java.util`, temos a classe `Stack` que estende `Vector`, expondo métodos desta que não são características de uma pilha.

Mas a motivação inicial dos criadores de `Properties` é extremamente válida: o reaproveitamento de código já escrito na classe `Hashtable`. Herança é comumente citada em cenários de reúso de código, mas não é a única solução. Usar composição permite reaproveitamento de código sem o efeito indesejado da quebra de encapsulamento. Para que a classe `Properties` reaproveitasse os recursos já existentes em `Hashtable` com composição bastaria um atributo `Hashtable` privado, e o método `setProperty` delegaria a invocação para o `hashtable.put`. Dessa maneira, expomos apenas os métodos que desejamos, sem possibilitar que fosse invocado o `hashtable.put` diretamente, nem que `Properties` pudesse ser passada como argumento para alguém que espera uma `Hashtable`. Melhor ainda, com o surgimento da API de Collections, como no Java 1.2, a implementação poderia ser facilmente trocada por algum `Map` sem causar problemas aos usuários da classe.

O uso abusivo de herança ocorre em diversos frameworks e APIs. O Struts 1 é um exemplo deste uso e de seus diversos problemas relacionados; somos instruídos a estender `Action` e, ao mesmo tempo, a tomar muito cuidado com seu ciclo de vida, principalmente ao sobrescrever alguns métodos-chave dessa classe. Em linguagens com suporte a *mixin*, como Ruby, o acoplamento com as classes pode gerar complicações ainda maiores, que só serão percebidas em tempo de execução. [176]

O problema da herança aparece, mesmo que mais sutilmente, no uso de classes abstratas para aplicar patterns, como o *template method*. Podemos reparar em alguns tropeços da implementação base de `Servlet`, `GenericServlet`, começando pelo método `init(ServletConfig)`:

```
public abstract class GenericServlet
                    implements Servlet, ServletConfig, Serializable {
    // ...

    public void init(ServletConfig config) throws ServletException {
        this.config = config;
        this.init();
    }
}
```

Ao criar nossa própria servlet na Web, somos obrigados a herdar de `HttpServlet`, que, por sua vez, herda de `GenericServlet`. Se quisermos reescrever o método `init(ServletConfig)` em nossa classe, devemos sempre nos lembrar de invocar o `super.init(ServletConfig)`, caso contrário a `ServletConfig` não será guardada como atributo na `HttpServlet`, fazendo o respectivo getter retornar `null`, quebrando muitos códigos, como a seguir:

```
public class LogServlet extends HttpServlet {
    public void init(ServletConfig cfg) throws ServletException {
        log("Inicializando a servlet");
    }

    public void doGet(HttpServletRequest req, HttpServletResponse res) {
        // vai lançar NullPointerException
        getServletConfig().getInitParameter("...");
    }
}
```

O uso de herança para criar uma filha de `HttpServlet` gera necessidade de saber como o método `init(ServletConfig)` foi implementado na classe mãe, o que consiste em uma quebra de encapsulamento.

Na mudança de API da Servlet 2.0 para a 2.1, houve a inserção de um método para tentar evitar este problema, que se tornara recorrente entre os desenvolvedores. O novo método `init()`, sem parâmetros, deve ser reescrito; ele é invocado pelo próprio `init(ServletConfig)` após o `ServletConfig` ser guardado em um atributo. Dentro de um `init()` reescrito podemos obter acesso ao `ServletConfig` através do *getter* da classe mãe, que agora estará corretamente atribuído. Mesmo assim, em razão da tomada de decisão original na API de servlet, o desenvolvedor ainda deve conhecer como esses métodos funcionam na classe mãe, pois só assim saberá de todos os cuidados que são necessários.

Mas nem tudo é negativo. Todo esse design de servlets tem como objetivo principal trazer o polimorfismo e uma interface de uso uniforme para as servlets serem chamadas pelo container. Contudo, em Java, é possível usar `interface` para polimorfismo, sem os pontos negativos da herança como quebra de encapsulamento e alto acoplamento. A API de servlets poderia ter usado um design baseado em interfaces com um *strategy pattern* por exemplo, evitando a herança. Uma sugestão seria uma interface `InvocationProcessor` com métodos como `init(ServletConfig)` e `process`:

```
public interface InvocationProcessor {
    void init(ServletConfig config);
    void process(HttpServletRequest req, HttpServletResponse res);
    boolean shouldProcess(HttpServletRequest request);
    void destroy();
}
```

Poderíamos criar várias implementações, cada uma capaz de lidar com as requisições que achar conveniente, como, por exemplo, para o método `GET`:

```
public class GetProcessor implements InvocationProcessor {
    public void init(ServletConfig config) { }

    public boolean shouldProcess(HttpServletRequest request) {
        return "GET".equals(request.getMethod());
    }

    public void process(HttpServletRequest req, HttpServletResponse res) {
        // código de processamento aqui
    }

    public void destroy(ServletConfig config) { }
}
```

Ao substituir a herança por interface, continuamos com o benefício do polimorfismo. Mas herança ainda traz outro grande benefício, o reaproveitamento de código, algo que interfaces puras não fazem. **Podemos então usar interfaces com composição, obtendo substitutos para todos os benefícios de herança sem correr o risco de cair na armadilha da quebra de encapsulamento e alto acoplamento**. Repare que, desta forma, o acoplamento é bem menor, nenhuma das classes precisa conhecer o funcionamento interno das outras.

Porém, se for necessário o uso de herança, alguns cuidados são importantes para minimizar a possível quebra de encapsulamento. Joshua Bloch, no *Effective Java*, fala de *Design for Inheritance*, ([17]) com diversas práticas como evitar invocações entre métodos públicos, para que a sobrescrita de um não mude o comportamento de outro.

Ao mesmo tempo que desejamos evitar herança, temos interesse em permitir mudança no nosso comportamento sem a necessidade de alterar ou copiar o código original. Mantendo pontos de extensão, permitimos que classes que implementam uma determinada interface sejam capazes de modificar o comportamento de outras

classes. Através de interfaces e composição, podemos criar desde um novo driver JDBC para um determinado banco de dados, ou flexibilizar trabalhos simples, como ordenar listas com `Comparators` implementando diferentes critérios de comparação. Este princípio, de manter suas classes abertas para extensão sem a necessidade de alteração no código original, é chamado *Open Closed Principle*.[124]

Evite herança, favoreça composição é um de dois princípios fundamentais de design do livro *Design Patterns*, com frequência referenciado na literatura. [79], [206]

3.4 Favoreça imutabilidade e simplicidade

Dois clientes com a mesma data de pagamento não podem compartilhar instâncias de `Calendar`, pois, se um deles alterar o vencimento, essa mudança resultaria em um efeito colateral provavelmente indesejado: o outro também sofre a alteração. A classe `Calendar` permite mudança no estado de suas instâncias e, portanto, diz-se que a classe é mutável, como o exemplo a seguir demonstra:

```
Calendar data = new GregorianCalendar(2004, 2, 13);
cliente.setVencimento(data);

Calendar doisDiasDepois = cliente.getVencimento();
doisDiasDepois.add(Calendar.DAY_OF_MONTH, 2);

outroCliente.setVencimento(doisDiasDepois);

// efeito colateral:
System.out.println(cliente.getVencimento());
```

A classe `String` tem comportamento diferente. Quando começamos a programar com Java, passamos por um código semelhante ao que segue:

```
String s = "java";
s.toUpperCase();
System.out.println(s);
```

Este código, que aparece com frequência nas provas de certificação, imprimirá `java` em letras minúsculas. Isso porque a classe `String` é dita imutável. Todo método invocado em uma `String`, que parece modificá-la, sempre devolve uma nova

instância com a alteração requisitada, mas nunca altera a instância original. Ao tornar uma classe imutável, uma ampla gama de problemas comuns desaparece. [82], [17]

Simplicidade e previsibilidade

Objetos imutáveis são muito mais simples de manipular que os mutáveis, tendo um comportamento bem previsível:

```
String s = "arquitetura";
removeCaracteres(s);
System.out.println(s);
```

Não importa o que o método `removeCaracteres(s)` faça, a saída apresentará *arquitetura*. Objetos imutáveis não sofrem efeitos colaterais, pois têm comportamento previsível em relação ao seu estado. Compare com um código semelhante, mas passando uma referência a um objeto mutável; no caso, um `Calendar`:

```
Calendar c = Calendar.getInstance();
verificaFeriado(c);
System.out.println(c.get(Calendar.YEAR));
```

Por mais explícito que seja o nome do método, não é possível afirmar, apenas com este trecho de código, qual é a saída no console, pois o método `verificaFeriado` pode alterar algum dado desta instância de `Calendar`. Os próprios engenheiros da Sun admitiram alguns erros de design das APIs antigas, como a classe `java.util.Calendar` ser mutável. [16] Por isso, muitas vezes recorremos às APIs de terceiros, como a **Joda Time**, nas quais encontramos entidades de datas, horários e intervalos imutáveis.

Quando o objeto é mutável, para evitar que alguém o modifique, temos que tomar alguma precaução. Uma solução, frequentemente usada com coleções através do `Collections.unmodifiableList(List)` e seus similares, é interfacear o objeto e embrulhá-lo de tal forma que os métodos expostos não possibilitem modificação. Em vez de passar a referência original adiante, passamos essa nova referência, cuja instância lança exceções em qualquer tentativa de modificação.

Outra solução para objetos mutáveis é criar **cópias defensivas** do objeto. Em vez de devolver o objeto original que se deseja preservar, criamos uma cópia, por exemplo, através do método `clone`, e devolvemos esta cópia. O usuário, desta forma,

não tem como alterar o objeto original, mas recebe uma cópia para uso próprio, que, se alterar, afetará apenas a si mesmo.

Objetos imutáveis são mais simples de se lidar. Depois de sua criação, sempre saberemos seu valor e não precisamos tomar cuidados adicionais para preservá-lo. Objetos mutáveis, em contrapartida, com frequência necessitam de um cuidado adicional, cujo objetivo é evitar alterações inesperadas.

Otimizações de memória

Podemos tirar proveito da imutabilidade de outras formas. [82] Como cada objeto representa apenas um estado, você não precisa mais do que uma instância para cada estado. A própria API da linguagem Java usa a imutabilidade como uma vantagem para fazer cache e reaproveitamento de instâncias.

É o caso do *pool* de `Strings` da JVM, que faz com que `Strings` de mesmo conteúdo possam ser representadas por uma única instância compartilhada. O mesmo acontece em boa parte das implementações das classes wrapper como `Integer`. Ao invocar `Integer.valueOf(int)`, a referência a `Integer` devolvida pode ser fruto de um cache interno de objetos com números mais frequentemente solicitados. [41]

Esse tipo de otimização só é possível com objetos imutáveis. É seguro compartilhar instâncias de `Strings` ou `Integers` com diferentes partes do programa, sem o risco de que uma alteração no objeto traga efeitos colaterais indesejados em outras partes do sistema.

E há mais possibilidades ainda para otimizações graças à imutabilidade. A classe `String` ainda se aproveita dessa característica para compartilhar seu array de `char` interno entre `Strings` diferentes. Se olharmos seu código fonte, veremos que ela possui três atributos principais: um `char[] value` e dois inteiros, `count` e `offset`; [133] estes inteiros servem para indicar o início e o fim da `String` dentro do array de `char`

Seu método `substring(int,int)` leva em conta a imutabilidade das `Strings` e os dois inteiros que controlam início e fim para reaproveitar o array de `char`. Quando pedimos uma determinada *substring*, em vez de o Java criar um novo array de `char` com o pedaço em questão, ele devolve um novo objeto `String`, que internamente compartilha a referência para o mesmo array de `char` que a `String` original, e tem apenas os seus dois índices ajustados. Ou seja, criar uma substring em Java é muito leve, sendo apenas uma questão de ajustar dois números inteiros (Figura 3.3).
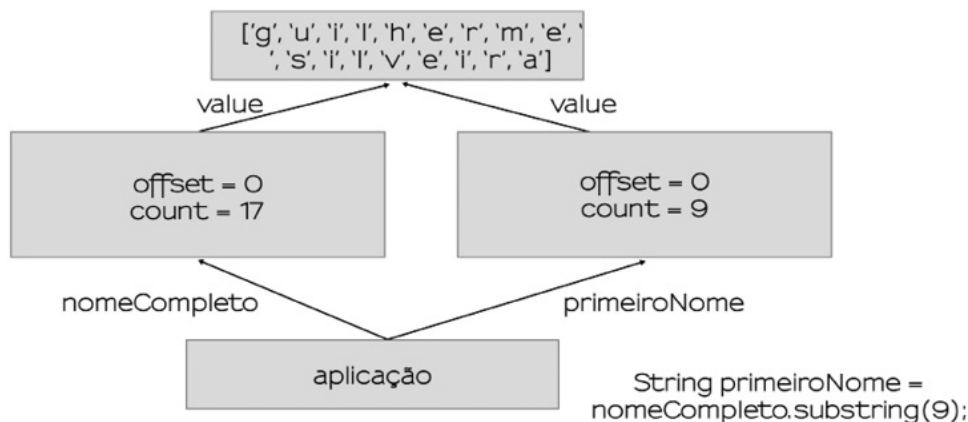
Figura 3.3: Compartilhando o mesmo array entre duas Strings distintas.

Essa otimização é uma implementação um pouco mais simples do design pattern *flyweight*, em que se propõe reaproveitar objetos com objetivo de diminuir o uso da memória. [17] O próprio pool de `Strings` pode ser considerado um *flyweight*. E poderíamos ir mais longe com o *flyweight*, implementando a concatenação de Strings apontando para diversas outras Strings internamente ao invés de copiar os dados para seu próprio array de char. Neste caso, a API do Java prefere não chegar a este ponto, pois, apesar de um ganho de memória, haveria o *trade-off* do aumento de processamento.

É válido também ressaltar o perigo de certas otimizações em alguns casos. Na otimização do substring, por exemplo, uma `String` pequena pode acabar segurando referência para um array de `chars` muito grande se ela foi originada a partir de uma `String` longa. Isso impediria que o array maior fosse coletado, mesmo se não possuirmos referências para a `String` original.

Há também um excesso de memória consumida temporariamente. Com imutabilidade, cada invocação de método cria uma nova cópia do objeto apenas com alguma alteração e, se isto estiver dentro de um laço, diversas cópias temporárias serão utilizadas, mas apenas o resultado final é guardado. Aqui, o uso do pattern *builder* pode ajudar, como é o caso da classe mutável `StringBuilder` (ou sua versão thread safe `StringBuffer`) com seu método `append` em vez de `String.concat` (ou o operador sobrecarregado +) em um laço.

Acesso por várias threads

Definir regiões críticas com `synchronized` é ainda uma solução repetidamente encontrada ao manipular memória compartilhada. Desta forma, evitamos que duas escritas concorrentes se entrelacem e que leituras sejam capazes de acessar dados inconsistentes ou intermediários.

A tarefa difícil é definir todas as regiões críticas e quais são mutuamente exclusivas; definir uma região muito grande gera perda de vazão (*throughtput*), enquanto uma de tamanho insuficiente implicará os mesmos problemas de não tê-las.

Em vez de recorrer aos recursos das linguagens, pensemos nesses objetos de forma diferente, evitando esse estado intermediário, não permitindo a mudança de valores. A vantagem é a *thread-safety*, pois, como não existem estados intermediários, não há como acessar nem modificar dados em momentos de inconsistência. O estado inconsistente fica escondido na lógica de construção, e o novo estado só estará disponível quando terminar de ser construído, para então ter sua referência compartilhada por mais de uma thread.

As linguagens funcionais mais puras, como *Erlang* e *Haskell*, estão sendo utilizadas em ambientes de grande concorrência, dada sua característica de trabalhar quase que apenas com valores imutáveis. Perde-se o conceito de *variável* como o conhecemos, já que os valores não mudam; nascem e morrem com o mesmo estado. Você é obrigado a programar apenas de maneira imutável, o que é uma enorme mudança de paradigma. Muitas dessas linguagens podem rodar sobre a JVM, como Scala, que, apesar de suportar mutabilidade, tem fortes raízes em programação funcional e recomenda o uso de imutabilidade.

Imutabilidade e encadeamento de invocações de métodos

Uma vez que as classes são imutáveis, os métodos não possuem efeitos colaterais e, portanto, devem devolver uma referência a um novo objeto (Figura 3.4). Logo, é natural encadear invocações de métodos (*method chaining*), viabilizando a criação de código mais legível:

```
Data vencimento = new Data();
Calendar proximaParcela = vencimento.adicionaMes(1)
                                    .proximoDiaUtil()
                                    .toCalendar();
```
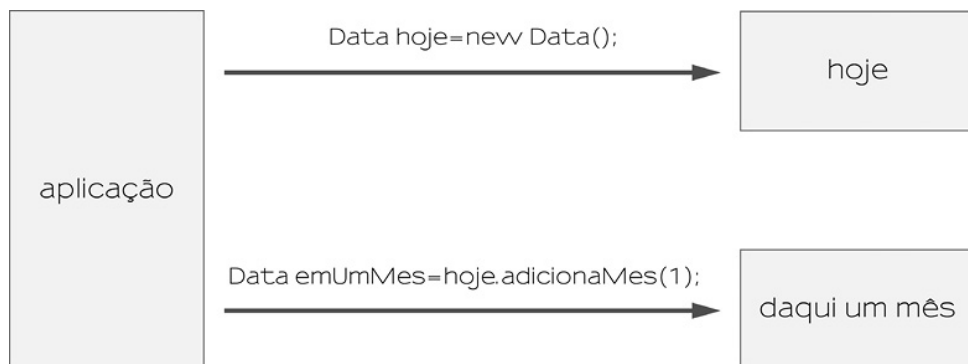
Figura 3.4: Cada invocação de método que cria uma situação nova, cria um estado novo, devolve a referência para um objeto novo.

O mesmo acontece com outras classes imutáveis do Java, como `BigDecimal`, `BigInteger` e `String`:

```
String valor = "arquiteturajava.com.br";
String resultado = valor.toUpperCase().trim().substring(6);
```

Method chaining é uma prática simples que permite a criação de interfaces fluentes (*fluent interfaces*) ([72]) que, junto com outros padrões e aplicada a domínios específicos, permite a criação de *DSLs*.

Criando uma classe imutável

Para que uma classe seja imutável, ela precisa atender a algumas características:[17]

- Nenhum método pode modificar seu estado.
- Ela deve ser `final`, para evitar que filhas permitam mutabilidade.
- Os atributos devem ser privados.
- Os atributos devem ser `final`.
- Caso sua classe tenha composições com objetos mutáveis, eles devem ter acesso exclusivo pela sua classe, devolvendo cópias defensivas quando necessário.

Uma classe imutável ficaria como:

```
public final class Ano {
    private final int ano;
    public Ano(int ano) {
        this.ano = ano;
    }

    public int getAno() {
        return this.ano;
    }
}
```

Este código seria mais complicado se nossa classe imutável fosse composta de objetos mutáveis. Uma classe `Periodo` imutável, implementada com `Calendars`, precisará trabalhar com cópias defensivas em dois momentos. Primeiro, quando receber algum `Calendar` como parâmetro e, depois, quando devolver algum `Calendar` que faça parte da composição do objeto. Se não copiarmos os objetos, é possível que algum código externo à classe `Periodo` altere os `Calendars` em questão, gerando inconsistência.

Note as cópias defensivas no construtor e nos getters no seguinte código:

```
public final class Periodo {

    private final Calendar inicio;
    private final Calendar fim;

    public Periodo(Calendar inicio, Calendar fim) {
        this.inicio = (Calendar) inicio.clone();
        this.fim = (Calendar) fim.clone();
    }

    public Calendar getInicio() {
        return (Calendar) inicio.clone();
    }

    public Calendar getFim() {
        return (Calendar) fim.clone();
    }
}
```

Aproveitamos aqui o fato de `Calendar` implementar `Cloneable`, caso contrário precisaríamos fazer a cópia manualmente, criando um novo objeto e alterando os atributos pertinentes um a um.

Como nossa classe é imutável, se precisarmos calcular alguma informação que exija qualquer modificação, clonamos o objeto. É o caso de um método que adie o período em uma semana, ou seja, some sete dias ao fim do período:

```
public Periodo adiaUmaSemana() {
    Calendar novoFim = (Calendar) this.fim.clone();
    novoFim.add(Calendar.DAY_OF_MONTH, 7);
    return new Periodo(inicio, novoFim);
}
```

E, com uma pequena modificação, podemos implementar o design pattern *flyweight* em nossa classe, compartilhando a instância do `Calendar` de início do, período entre o objeto original e o novo, com uma semana adiada. Para tanto, precisaríamos de um outro construtor privado para ser chamado no `adiaUmaSemana` que não fizesse o `clone`.

Utilizar classes imutáveis traz um trabalho a mais junto com os diversos benefícios descritos. **Você deve considerar fortemente criar sua classe como imutável.**

3.5 Cuidado com o modelo anêmico

Um dos conceitos fundamentais da orientação a objetos é o de que você não deve expor seus detalhes de implementação. Encapsulando a implementação, podemos trocá-la com facilidade, já que não existe outro código dependendo desses detalhes, e o usuário só pode acessar seu objeto através do contrato definido pela sua interface pública. [21]

Costumeiramente, aprendemos que o primeiro passo nessa direção é declarar todos seus atributos como `private`:

```
public class Conta {
    private double limite;
    private double saldo;
}
```

Para acessar esses atributos, um desenvolvedor que ainda esteja aprendendo vai rapidamente cair em algum tutorial, que sugere a criação de *getters* e *setters* para poder trabalhar com esses atributos:

```
public class Conta {
    private double limite;
    private double saldo;

    public double getSaldo() {
        return saldo;
    }

    public void setSaldo(double saldo) {
        this.saldo = saldo;
    }

    public double getLimite() {
        return limite;
    }

    public void setLimite(double limite) {
        this.limite = limite;
    }
}
```

Essa classe contém alguns métodos que acabam por ferir as ideias do encapsulamento. O método `setSaldo` é um bom exemplo disso, já que dificilmente o saldo em uma entidade `Conta` será simplesmente "substituído" por outro. Para alterar o saldo de uma conta, é necessário alguma operação que faça mais sentido para o domínio, como *saques* ou *depósitos*.

Nunca crie um getter ou setter sem uma necessidade real; lembre-se de que precisamos que essa necessidade seja clara para criar qualquer método que colocamos em uma classe. Particularmente, os *getters* e *setters* são campeões quando falamos em métodos que acabam nunca sendo invocados e, além disso, grande parte dos utilizados poderia ser substituída por métodos de negócio. [100]

Essa prática foi incentivada nos primórdios do AWT, para o qual era recomendado criar *getters* e *setters* para serem invocados no preenchimento de cada campo visual da sua interface gráfica com o usuário, cunhando o termo *JavaBean*. Os EJBs também contribuíram para esta prática, como será visto adiante.

Criando classes desta forma, isto é, adicionando *getters* e *setters* sem ser criterioso, códigos como `conta.setSaldo(conta.getSaldo() + 100)` estarão espalhados por toda a aplicação. Se for preciso, por exemplo, que uma taxa seja debitada toda vez que um depósito é realizado, será necessário percorrer todo o código e modificar

essas diversas invocações. Um *search/replace* ocorreria aqui; péssimo sinal. Podemos tentar contornar isso e pensar em criar uma classe responsável por esta lógica:

```
public class Banco {
    public void deposita(Conta conta, double valor) {
        conta.setSaldo(conta.getSaldo() + valor);
    }

    public void saca(Conta conta, double valor) {
        if (conta.getSaldo() >= valor) {
            conta.setSaldo(conta.getSaldo() - valor);
        } else {
            throw new SaldoInsuficienteException();
        }
    }
}
```

Esse tipo de classe tem uma característica bem procedural, fortemente sinalizada pela ausência de atributos e excesso do uso de métodos como funções (`deposita` e `saca` poderiam ser estáticos). Além disso, pode-se dizer que esta classe tem uma *intimidade inapropriada* com a classe `Conta`, pois conhece demais sua implementação interna. Repare que o método `saca` verifica primeiro se o saldo é maior que o valor a ser sacado, para, então, retirar o dinheiro. Esse tipo de lógica deveria estar dentro da própria classe `Conta`.

O princípio do *Tell, Don't Ask* prega exatamente isso: você deve dizer aos objetos o que fazer (como sacar dinheiro), evitando perguntar em excesso o estado ao objeto, como `getSaldo`, e, a partir desse estado, tomar uma decisão. [172]

Esse tipo de classe é comumente encontrada e é classificada como o pattern *Business Object* por concentrar a lógica de negócios. Já a classe `Conta`, por ter apenas os dados, recebia o nome *Value Object* (hoje, este pattern tem outro significado). Da forma como está, temos separados nossos dados na classe `Conta` e a lógica de negócio na classe `Banco`, rompendo o princípio básico de manter comportamento e estado relacionados em uma única classe.

É o que chamamos de **modelo anêmico** (*anemic domain model*), [77], no qual nossa classe `Conta` parece não ter responsabilidade alguma no sistema, nenhuma ação relacionada. É necessário uma classe externa, `Banco`, para dar alguma ação para nossa `Conta`, tratando-a quase como um fantoche. [29] Com isso, a classe `Banco` conhece detalhes da implementação da classe `Conta` e, se esta mudar, `Banco` muito provavelmente mudará junto.

Podemos unir a lógica de negócio aos dados de uma maneira simples, inserindo métodos na classe `Conta` e removendo os que apenas acessam e modificam diretamente seus atributos:

```
public class Conta {
    private double saldo;
    private double limite;

    public Conta(double limite) {
        this.limite = limite;
    }

    public void deposita(double valor) {
        this.saldo += valor;
    }

    public void saca(double valor) {
        if (this.saldo + this.limite >= valor) {
            this.saldo -= valor;
        } else {
            throw new SaldoInsuficienteException();
        }
    }

    public double getSaldo() {
        return this.saldo;
    }
}
```

Aqui mantivemos o `getSaldo`, pois faz parte do domínio. Também adicionamos algumas manipulações ao método `saca`, e poderíamos debitar algum imposto em cima de qualquer movimentação financeira no método `deposita`. **Enriqueça suas classes com métodos de negócio, para que não se tornem apenas estruturas de dados**. Para isso, cuidado ao colocar *getters* e *setters* indiscriminadamente. Devemos encapsular os dados em atributos de objetos e, ainda, lembrar que a orientação a objetos prega a troca de mensagens (invocação de métodos) de maneira a concentrar as responsabilidades a quem pertence os dados. O próprio Alan Kay, que cunhou o termo "programação orientada a objetos", ressalta que "*o termo foi uma má escolha, pois diminui a ênfase da ideia mais importante, a troca de mensagens*". [35]

É possível seguir a mesma linha para entidades do JPA/Hibernate, verificando

a real necessidade dos getters e setters. Por exemplo, a necessidade de um método `setId` para a chave primária torna-se discutível no momento em que um framework utiliza reflection ou manipulação de bytecode para ler atributos privados.

Algumas vezes, os *getters* e *setters* são, sim, necessários, e alguns patterns até mesmo precisam de uma separação de lógica de negócios dos respectivos dados. [73] Práticas como o *Test Driven Development* podem ajudar a não criar métodos sem necessidade.

Porém, frequentemente, entidades sem lógica de negócio, com comportamentos codificados isoladamente nos *business objects*, caracterizam um modelo de domínio anêmico. É muito fácil terminar colocando a lógica de negócio, que poderia estar em em nossas entidades, diretamente em Actions do Struts, `ActionListeners` do Swing e *managed beans* do JSF, transformando-os em *transaction scripts*. Este modelo acaba ficando com um forte apelo procedural e vai diretamente na contramão das boas práticas de orientação a objetos e do *Domain-Driven Design*. [21], [57]

3.6 Considere Domain-Driven Design

Todo software é desenvolvido com um propósito concreto, para resolver problemas reais que acontecem com pessoas reais. Todos os conceitos ao redor do *problema* a ser resolvido são o que denominamos domínio. O objetivo de toda aplicação é resolver as questões de um determinado domínio.

Domain-Driven Design (DDD) significa guiar o processo de design da sua aplicação pelo domínio. Parece óbvio, mas muitos softwares não são projetados de acordo com o domínio em que atuam. Podemos perceber essa realidade analisando o código de diversos sistemas atuais, nos quais as entidades não condizem com a realidade dos usuários e são de difícil entendimento. [57], [77]

Segundo o DDD, é impossível resolver o problema no domínio do cliente sem entendê-lo profundamente. É claro que o desenvolvedor não quer se tornar um completo especialista na área do cliente, mas deve compreendê-la o suficiente para desenvolver *guiado pelo domínio*.

Para isto acontecer, o ponto-chave é a **conversa**. Conversa constante e profunda entre os especialistas de domínio e os desenvolvedores. Aqueles que conhecem o domínio em detalhes devem transmitir conhecimento aos desenvolvedores. Juntos, chegarão a termos e vocábulos em comum, uma *língua comum*, que todos utilizem. É a chamada **Língua Ubíqua** (*Ubiquitous Language*).

Esta língua é baseada nos termos do domínio, não totalmente aprofundada neste,

mas o suficiente para descrever os problemas de maneira sucinta e completa.

Durante a conversa constante, cria-se um **modelo do domínio** (ou *Domain Model*). É uma abstração do problema real, que envolve os aspectos do domínio que devem ser expressados no sistema, desenvolvida em parceria pelos especialistas do domínio e desenvolvedores. É este modelo que os desenvolvedores implementam em código, que deve ocorrer literalmente, item por item, como foi acordado por todos. Isto possibilita o desenvolvimento de um código mais claro, e, principalmente, que utiliza metáforas próprias do domínio em questão.

Seu programa deve expressar a riqueza do *domain model*. Qualquer mudança no modelo deve ser refletida no código. Caso o modelo se torne inviável para se implementar tecnicamente, ele deve ser mudado para se tornar implementável.

Esse processo não ocorre antes do início do projeto, mas é um trabalho constante, que deve ser revisitado a todo instante. Praticando DDD, é comum encontrar situações em que o especialista e os desenvolvedores aprendem mais sobre o domínio e refatoram o código e suas entidades à medida que o projeto é desenvolvido e, por isso, práticas de *Test Driven Design* e refatoração são complementares a DDD.

No DDD, seu código sempre será a expressão do modelo, que, por sua vez, é guiado pelo domínio.

A principal literatura sobre Domain-Driven Design é o livro homônimo de Eric Evans.<span class="char-style-override-4”>26 Nele, o autor apresenta o que é o DDD, a língua ubíqua e parte para um *catálogo de design patterns*. Esses *patterns* citados por Evans são ferramentas que facilitam a implementação do *Domain Model* no software. Eles são excessivamente referenciados em artigos, mas não são o ponto principal do DDD.

Em muitas discussões em torno do assunto, com frequência encontram-se receitas mágicas que, aparentemente, podem ser aplicadas em qualquer projeto. Este é um dos maiores erros que se pode cometer. Não existe receita pronta, e tampouco apenas uma única maneira de escrever suas classes. Se deseja usar DDD, lembre-se do principal: o domínio. Você pode criar um *Model* riquíssimo e um design elegante; mas, se ele não for uma expressão fiel do *Domain*, se não for a partir da língua comum, você não está usando DDD e o modelo sofrerá com a sua diferença com o domínio.

As quatro camadas do DDD

Evans propõe em seu livro a divisão da aplicação em quatro camadas (*layers*), para favorecer o isolamento do domínio, o encapsulamento, a flexibilidade e facilitar

o DDD (Figura 3.5).

A interface e a interação com o usuário (Swing, HTML, etc.) são a **UI Layer**. A **Application Layer** é que auxilia o domínio a ser executado, entregando e traduzindo as informações de que o domínio precisa. O principal ponto do DDD é o modelo do domínio e suas lógicas de negócio, que formam o **Domain Layer**. Por fim, a **Infrastructure Layer** contém serviços de infraestrutura necessários ao funcionamento das outras layers, persistência, segurança, envio de e-mail, entre outros (Figura 3.5).

Na implementação da *Domain Layer*, surgem necessidades mais ortogonais ou serviços mais complicados que fogem do domínio em questão, como persistência e comunicação SMTP para envio de e-mails. É papel da *Infrastructure Layer* servir a *Domain Layer* e às outras camadas com essas facilidades.
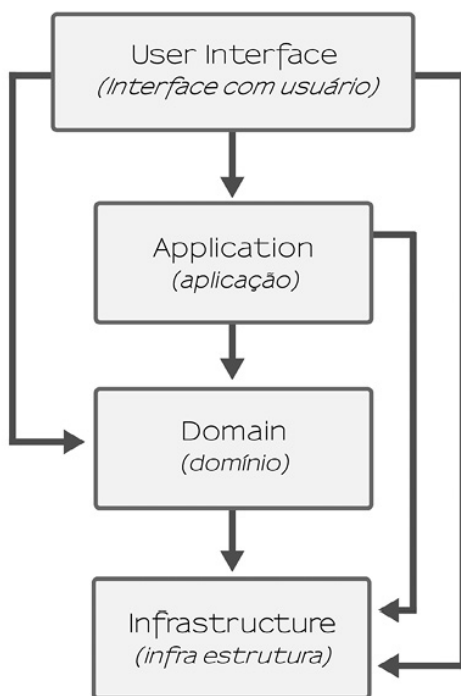
Figura 3.5: As quatro camadas do DDD.

A *UI Layer*, comumente chamada *view*, é responsável pela interface entre o usuário e seu sistema. É ela que interage com o usuário, recebendo suas ações e exibindo uma representação do modelo. *Application Layer* é uma camada fina que faz a ponte

entre a interface e o modelo do domínio. É ela que coordena e estimula o modelo de forma a atender à requisição da UI. Na maioria dos sistemas, é uma camada bem simples, e muitas vezes inexistente; se você usa um bom framework controlador, é bem capaz que ele já consiga fazer essa ponte entre UI e modelo automaticamente.

O DDD é todo sobre a *Domain Layer*. É nele que está a questão da Ubiquitous Language e do domain model. E nessa mesma camada entram os patterns que Evans cita, os quais são frequentemente discutidos na comunidade.

Há muitos patterns apresentados por Evans: alguns novos, outros de *model-driven design* e até alguns do GoF revisitados. Um dos mais simples é o **Value Object**, que modela um objeto geralmente imutável onde o que realmente interessa é seu valor. São objetos como `java.util.Calendar` ou `java.awt.Color`, em que não há identidade, e dois objetos com mesmo valor são iguais perante o sistema (um lugar que tenha um objeto `Calendar` com a data de hoje, pode ser trocado por um outro objeto `Calendar` com a mesma data).

Há ainda o pattern *Entity* (Entidade), que trata de objetos com identidade única, e ciclo de vida que queremos controlar. Em uma loja virtual, por exemplo, um `Produto` pode ser uma *entity*. Dois produtos com mesmo nome são coisas diferentes, e não podem ser trocados (há identidade em cada objeto, não é apenas o valor que conta). O ciclo de vida também é bem definido. O objeto é criado em um certo momento de cadastro, sua representação pode ser persistida e restaurada de um banco de dados, e ele é jogado fora quando for removido.

Sobre o ciclo de vida das *entities*, Evans propõe ainda uma série de patterns. Desde a versão DDD para *Factory*, aplicada à camada de domínio, até o famoso e mal-entendido *Repository*. Este é um padrão interessante para discutir vários conceitos do DDD e enxergar seus principais pontos em relação à modelagem do domínio.

Repositórios

Muitos domínios precisam persistir dados. O *Domain expert* (o especialista do domínio) pode não saber o que é um banco de dados nem o significado de *persistência*, mas sabe que quer *cadastrar* os clientes de sua empresa para depois *procurar* nesse cadastro. O cliente diz para o desenvolvedor (a conversa) que precisa *controlar* seu estoque de produtos para depois *buscar* quais produtos estão em baixa. O usuário tem, intuitivamente, o conceito de persistência e acesso das informações.

Já o desenvolvedor, conhece tudo de bancos de dados, integridade relacional, Hibernate, relacionamentos, SQL, normalização e qualquer outro detalhe técnico

envolvido. Nada disso é pertinente ao domínio do cliente, embora sejam todos conceitos fundamentais para se implementar a noção de persistência que o usuário tem na cabeça. É aí que entra o *Repository*.

Segundo Evans, repositório é um nome que aparece na língua ubíqua para representar aquele conceito rudimentar de persistência que existe na cabeça do cliente. É no repositório que cadastramos as informações. É nele que vamos, depois, procurar objetos já cadastrados. Por meio dele, controlamos os objetos e buscamos informações sobre eles.

A implementação em si do repositório não é foco do DDD. O desenvolvedor provavelmente criará um DAO com JPA ou JDBC, ou uma implementação direta de uma interface, mas isso foge do escopo do domain model. Escrever SQL e código de manipulação de banco de dados não faz parte da *Domain Layer*, a menos que seu domínio seja bancos de dados. Tudo isso faz parte da camada de infraestrutura, lugar onde seu DAO deve estar. A camada de domain deve apenas pensar em um repositório de objetos no qual as operações de persistência acontecem.

Os desenvolvedores conhecem o padrão DAO como uma forma de encapsular as particularidades do acesso a banco de dados (ou a alguma base de dados) e isolar essa complexidade do restante do programa. Agora, o DDD propõe o repositório, que representa o lugar onde nossos objetos são colocados para que depois possamos recuperá-los. E, infelizmente, muitos concluem que Repositório e DAO são o mesmo. Não são, embora seja fato que isso tudo gera muita confusão.

Repositório é um conceito do *Domain Layer*, que tem de fazer parte do *Modelo*. Note a diferença: DAO surge do problema de encapsular funcionalidades específicas de infraestrutura; o Repositório, da necessidade do cliente de guardar e obter objetos do domínio.

O nome do repositório não deve ser algo interno ao código, mas deve fazer parte da língua ubíqua; deve aparecer nas conversas e no Domain Model. Deve ser um conceito que o especialista de domínio também entenda e, porque está no *Model*, é que ele vai para o código. Não há problema em trazer palavreado técnico para a *Ubiquitous Language*, desde que todos entendam o significado, mantendo o princípio da língua ubíqua.

O padrão DAO não faz parte da língua ubíqua, mas da camada de infraestrutura. Existe uma relação entre Repositórios e DAOs, pois geralmente as buscas são feitas no banco de dados, e costuma ser boa prática ter um DAO para tanto, mas o DDD mesmo não diz nada sobre o assunto, já que isto está fora do domínio.

Segundo o DDD, há várias implementações possíveis, todas boas, desde que você

siga os conceitos discutidos. Uma encontrada com frequência é representar o Repositório na camada do domínio através de uma interface Java implementada direto na camada de infraestrutura. Em casos mais complexos, o repositório pode ser uma classe concreta que delega chamadas para mais de um DAO na infraestrutura. Mas isto não é regra. Desenvolva pensando no domínio, e temos DDD. Poderíamos ter:

```
public interface ClientesRepository {
    List<Cliente> getPorNome(String nome);
    List<Boleto> getPorCliente(Cliente cliente);
}
```

Seria muito mais conveniente ter a interface e seus métodos com nomes mais orientados a domínio, [31] como `Clientes` para a interface, ou ainda:

```
public interface TodosClientes {
    List<Cliente> comNome(String nome);
}

public interface TodosBoletos {
    List<Boleto> doCliente(Cliente cliente);
}
```

O nome `TodosClientes` ajuda no entendimento do que é um repositório; tecnicamente, ele tenta simular uma coleção em memória, mas na verdade é persistente. Esta característica ajuda o cliente a entender o que o repositório faz durante uma conversa com os desenvolvedores. Alguns detalhes de que devemos nos lembrar ao usar os repositórios são evitar ter métodos com nomes técnicos e agrupar as funcionalidades por tipo; se um método retorna uma lista de boletos, ele provavelmente pertence ao repositório de boletos.

Domain-Specific Language

Aproveitando a discussão sobre modelo e domínio, percebemos que, ao criar um código baseado no modelo, ele fica delimitado pelos elementos que forem abordados no escopo do domínio, e, consequentemente, na língua ubíqua.

Essas classes e os métodos da aplicação serão específicos para um determinado problema; neste caso, então, acabamos criando uma biblioteca específica para o domínio da aplicação, e podemos considerar que esta biblioteca é uma **DSL**, ou *Domain-Specific Language*.

Se considerarmos que tudo o que foi criado usa uma linguagem hospedeira, por exemplo Java ou Ruby, estamos falando de *DSLs* internas.

O código a seguir mostra um exemplo em Java de DSL interna:

```
cliente.fazReserva(paraQuarto(numeroDoQuarto)).
de(dataDeInicio).ate(dataDeSaida);
```

Podemos encarar esse trecho de código como uma nova linguagem. Mas perceba que ele ainda respeita as regras de sintaxe do Java, sua linguagem hospedeira. É código Java válido. O uso de uma linguagem hospedeira é o que caracteriza uma *DSL* interna.

Essas *DSLs* beneficiam-se de funcionalidades que sua linguagem possui. Para projetar e criar uma *DSL* interna, podemos usar técnicas de programação que a linguagem hospedeira permite, como o encadeamento de métodos (*Method Chaining*) do nosso exemplo.

Isso torna a construção de *DSLs* internas muito mais simples. Diversas vezes, essas *DSLs* surgem naturalmente ao refatorar o código para que expresse cada vez mais o domínio e a língua ubíqua. Encadeamento de métodos, *varargs*, *static imports*, *generics*, polimorfismo, classes anônimas e proxies dinâmicos são algumas das funcionalidades aplicadas na construção de *DSLs* internas em Java. As *DSLs* são criadas em cima de um modelo que representa o domínio, um modelo semântico *(semantic model*). [62]

Repare nos exemplos a seguir uma DSL de um repositório, que permite a composição de uma tarefa a partir de partes menores:

```
List<Quarto> vagas = quartos.vagosNoPeriodo(proximaSemanaUtil())
                .dosTipos(Tipos.PADRAO, Tipos.LUXO)
                .lista();

List<Cliente> hospedes = clientes.queSeHospedaram(data)
                    .nos(quartos.doTipo(Tipos.PADRAO))
                    .lista();
```

Além disso, nada nos impede de, em vez de usar uma DSL hospedada em uma *GPL* (*General Purpose Language*, ou Linguagem de Propósito Geral), criar uma nova sintaxe e linguagem para resolver o problema. Chamamos essas linguagens de DSL externa. Exemplo:

```
cliente faz reserva do quarto 215 de 12/02/1990 ate 20/02/1990
```

A grande vantagem de criar uma *DSL* externa é a possibilidade de restringir seu vocabulário, dexando-a bem próxima da língua ubíqua. Exemplos de DSLs externas são SQL, CSS e expressão regular. [62] Outro aspecto importante é que a *DSL* externa também não precisa respeitar as regras de sintaxe de nenhuma linguagem hospedeira, o que dá uma enorme flexibilidade ao projetá-la.

Por outro lado, a desvantagem de *DSLs* externas é a implementação de interpretadores, *parsers* e/ou compiladores. Já que criamos uma nova sintaxe, é muito trabalhoso conceber um interpretador, parser ou compilador do zero. *Parser generators*, também conhecidos como compiladores de compiladores (*compiler compilers*), ajudam bastante. Mesmo assim, ainda não é uma tarefa fácil.

Há alguns anos, tinha-se a ideia de que a criação de DSLs serviria para que, através delas, pessoas que não são programadores conseguissem escrever suas próprias rotinas. Mas, após algum tempo de discussão sobre o assunto, essa ideia perdeu força e, atualmente, pensa-se que as DSLs devam ser escritas por programadores, mas continuem fáceis de ler por pessoas que não programam, como é o caso dos business experts. [74]

Ao trabalhar com *DSLs*, considere o uso de frameworks que auxiliem na sua criação. Um dos frameworks Java mais usados para criar *DSLs* externas é o ANTLR, um *parser generator* usado em importantes projetos, como na interpretação de *HQL* (*Hibernate Query Language*). Em Ruby, há o Gherkin, usado pelo Cucumber na interpretação de suas especificações executáveis. O Hamcrest, usado no JMock, ajuda bastante na criação de *DSLs* internas em Java, além de permitir a criação de componentes reutilizáveis entre *DSLs* distintas.

DSLs podem ser escritas com ênfase em qualquer domínio, trazendo para o desenvolvimento do software a capacidade de exprimir um comportamento, através da linguagem mais próxima da realidade do problema a ser resolvido, e tentando remover ao máximo o ruído sintático da biblioteca, linguagem ou padrões utilizados para suportá-la. A maior parte do código passa a ser escrita com o vocabulário do domínio, e não mais com o da ferramenta.

Capítulo 4

Separação de responsabilidades

Ao pensar no design de uma aplicação, o foco não deve ser apenas na relação entre dois objetos, mas também entre grupos de objetos que trabalham em conjunto para uma determinada finalidade. Ao desenhar pequenas classes que, juntas, realizam uma tarefa maior, o programador ganha em simplicidade e modularidade.

É mais simples manter um sistema composto por classes que contêm apenas uma responsabilidade bem definida, já que a alteração de um comportamento é feito em uma única classe do sistema. Por esse e outros motivos, o programador deve buscar classes com baixo acoplamento e alta coesão, evitando padrões conhecidos que levam a designs rígidos e de alto custo de manutenção.

4.1 Obtenha baixo acoplamento e alta coesão

Para realizar alterações com facilidade em um sistema, queremos que mudanças em uma parte dele não impliquem mais mudanças em outras. A necessidade de atualizações em diversos pontos do código no momento de realizar alguma alteração de

funcionalidade pode indicar que esses pontos estão ligados (ou acoplados) de maneira indesejada. **Acoplamento** é o quanto dois elementos estão amarrados entre si e quanto as alterações no comportamento de um afetam o de outro.

Pode-se falar do acoplamento entre duas classes, ou do quanto dois módulos da aplicação estão amarrados, ou, ainda, avaliar o quanto dois frameworks distintos são afetados um pelo outro. Não se quer, por exemplo, alterar uma classe `Usuario` porque houve mudanças na regra da `ControleDeAcesso`, ou ter de alterar DAOs por causa de mudanças na View, ou ainda ser impedido de usar algum framework Web porque adotamos Hibernate para persistência.

Partes do sistema podem ser chamadas de componentes. Classes individuais, conjunto de classes, um pacote ou até mesmo uma biblioteca contida em um arquivo JAR são exemplos de componentes. Em todos os casos, minimizar o acoplamento implica também facilitar a troca dos mesmos, além de ser peça fundamental para a manutenção do código, formando uma arquitetura de maior qualidade.

Mas *baixo* acoplamento é diferente de *nenhum* acoplamento. Para que dois componentes se comuniquem, sempre haverá uma ligação. Ao buscar facilidade de manutenção a longo prazo, devemos lutar para que essa ligação seja a menor e mais simples possível. Descobrir, entender e gerenciar essas dependências é uma das partes mais complicadas do processo de desenvolvimento de software. Bob Martin diz que os designs deterioram à medida que novos requisitos forçam mudanças que não foram previstas, e isso faz com que sejam introduzidas dependências novas e não planejadas entre classes e módulos do sistema.[126] Fowler tem uma opinião parecida e diz que, à medida que os sistemas crescem, é necessária uma maior atenção ao gerenciamento dessas dependências, pois, caso contrário, simples alterações podem ser propagadas para outras classes ou módulos, prejudicando assim a evolução do software.[69]

Encapsulamento e bom uso de interfaces são essenciais para diminuir o acoplamento entre componentes, mas há ainda um ponto importante quando se pensa em um nível mais alto, olhando-se para o sistema como um todo: as **responsabilidades** de cada componente. A razão da existência de todo componente é exercer algum tipo de função no sistema, assumir certo requisito como sua responsabilidade. E uma importante e boa prática é garantir que ele tenha tenha **alta coesão**, o que significa garantir que suas responsabilidades sejam relacionadas e façam sentido juntas, ou seja, que não haja responsabilidades desconexas dentro de um mesmo componente. Isso facilita o entendimento e futura modificação do componente, além de aumentar suas chances de reutilização já que, em geral, poucas partes do sistema precisariam

de um componente com responsabilidades muito diversas.

Bob Martin definiu responsabilidade como "*uma razão para mudança*", em seu artigo "*Single Responsibility Principle*".[120] Nele, Martin defende que, além de ter alta coesão, o componente deve ter apenas uma única responsabilidade, "*em momento algum deve existir mais de um motivo para alterar uma classe*". Isso diminui o acoplamento entre responsabilidades diferentes, evitando que mudanças em uma afetem a outra. Nesse mesmo artigo, ele ainda cita: "*se você consegue pensar em mais de um motivo de mudança para sua classe, então ela tem mais de uma responsabilidade*".

Um conceito ainda mais amplo e difundido como excelente prática arquitetural é o da **Separação de Responsabilidades**, ou *SoC*, do inglês *Separation of Concerns*. Para diminuir o acoplamento, aumentar a coesão, promover a flexibilidade e garantir responsabilidades únicas, é preciso saber separar essas responsabilidades. Há quem diga que o termo apareceu pela primeira vez em 1974, nas palavras de Edsger Dijkstra, quando argumentava que, ao desenvolver um software, muitos aspectos diferentes devem ser tratados.[47] E, como ele diz: "*Nada se ganha – ao contrário! – abordando esses diversos aspectos simultaneamente. É o que eu algumas vezes chamei de 'separação de responsabilidades'*". E, mais à frente, explica: "*Isso não significa ignorar os outros aspectos, mas estar apenas fazendo justiça ao fato de que, sob o ponto de vista de um aspecto, o outro é irrelevante*".

Em um nível mais próximo ao código, um método que verifica permissões do usuário logado e monta um relatório, possui mais de uma responsabilidade diferente e requer refatoração. Aqui, separar significa criar dois métodos distintos, um para a autorização e outro para a execução da tarefa. Levando o pensamento a um nível superior, caso os métodos fiquem na mesma classe, ela continua com dois papéis distintos: autenticação e execução. O resultado seria separar o código entre duas classes. Ou, mais ainda, em pacotes ou submódulos diferentes da aplicação. E até, se for o caso, em frameworks especializados em cada tarefa.

Como se pode ver, esse trabalho pode ser feito em diversos níveis, e não só em linguagens orientadas a objeto. No mundo Web, arquivos HTML preocupam-se em descrever conteúdo, arquivos CSS descrevem a apresentação e, por fim, arquivos JavaScript, a lógica da sua página. É possível misturar todo o código, mas também podemos separá-lo em diversas partes, cada uma com sua própria responsabilidade.

Ao longo dos anos, SoC foi ganhando definições mais robustas e se tornou amplamente difundido. Em orientação a objetos, muito se fala do assunto para definir o papel de cada classe do sistema. Fala-se de SoC quando se organiza as camadas da

aplicação, na separação do *MVC*, na relação com inversão de controle, orientação a aspectos e em várias outras situações.

Há várias práticas e abordagens que facilitam a separação de responsabilidades, como veremos nos próximos tópicos.

4.2 Gerencie suas dependências através de injeção

Mesmo com o baixo acoplamento haverá sempre uma ligação entre duas classes que precisam trabalhar em conjunto. Normalmente, há uma classe que necessita dos serviços oferecidos por outra. Qualquer mudança nesta classe que está sendo acessada pode afetar o comportamento da primeira. Há aqui uma relação natural de dependência.[110] Relações como essa indicam ainda que a semântica do cliente é incompleta sem o fornecedor.[69]

Suponha uma classe chamada `CalculadoraDeSalario`, responsável por calcular o salário de um determinado funcionário. Para estimar quais impostos serão descontados do empregado, esta classe precisa da `TabelaDeImpostos`. Existe uma relação de dependência entre essas duas classes e, por isso, dizemos que a `CalculadoraDeSalario` **depende** da `TabelaDeImpostos`. Quando o assunto é organização de classes e objetos, as dependências entre as diversas partes do sistema são um assunto delicado. Devemos nos esforçar ao máximo para diminuir o acoplamento.

Um caso recorrente em diversos projetos é a tentativa de encapsular todo o acesso a dados (persistência) de uma aplicação, centralizando-o em objetos específicos, aplicando o pattern *Data Access Object* (DAO).

```
public class ProjetoDao {
    public void salva(Projeto projeto) {
        // ...
    }

    public void remove(Projeto projeto) {
        // ...
    }

    public Projeto carrega(Long id) {
        // ...
    }
}
```

Partindo do princípio de que essa classe utiliza diretamente JDBC para o acesso ao banco, ela precisa de uma conexão, uma referência a `java.sql.Connection`, em todos os seus métodos. O uso de algum framework para acesso a dados ou de mapeamento objeto-relacional não resolveria o problema, já que ela precisaria de algo análogo a uma `Session` ou `EntityManager`.

O problema é que o DAO foi criado para encapsular os detalhes de acesso aos dados, porém, precisa de alguns recursos (neste caso, uma conexão) para realizar seu trabalho. Seria o mesmo caso com um `EnviadorDeEmails`, que necessita de uma conexão com o SMTP, por exemplo.

Uma primeira possibilidade para obter uma referência para `Connection` seria abrir conexões diretamente nos métodos que necessitam delas, deixando os detalhes por conta do próprio DAO:

```
public class ProjetoDao {
    public void salva(Projeto projeto) throws SQLException {
        String url = "jdbc:mysql://localhost/db";
        String usuario = "root";
        String senha = "password";

        Connection connection =
        DriverManager.getConnection(url, usuario, senha);
        // ::uso da conexão ...::
        connection.close();
    }

    // ::outros métodos do DAO::

}
```

O código anterior serve para conectar em uma base do MySQL. É inviável repeti-lo em todos os outros lugares que precisem de conexão. Além disso, ainda existem diversos outros detalhes importantes a serem considerados, como a externalização das configurações e o uso de um **pool de conexões** para gerenciamento mais inteligente dos recursos.

Repare que a classe `ProjetoDao` tem responsabilidades demais; além de saber como salvar, buscar e alterar projetos, ainda é responsável por criar uma conexão com a base de dados. Ela é um típico exemplo de uma classe com baixa coesão, devido a tantas responsabilidades.

O primeiro passo para amenizar o problema é centralizar o processo de criar conexões. Dessa forma, a classe pode depender apenas de uma interface e ignorar completamente os detalhes de criação de conexões (se a conexão é nova, se veio de um pool, qual driver estamos usando, etc.). O uso de uma *factory* permite esconder e centralizar tais detalhes e trocar essa estratégia mais adiante, caso necessário.

```
public void salva(Projeto projeto) throws SQLException {
    Connection connection = new ConnectionFactory().getConnection();

    // ::uso da conexão:: ...

    connection.close();
}
```

A obtenção das conexões fica mais simples, mas ainda não resolve o problema. Com esse código, não é possível salvar dois `Projetos` utilizando a mesma conexão, nem executar dois métodos do DAO dentro de uma mesma transação; a cada nova invocação de método uma nova conexão é adquirida. Essa é uma má prática, conhecida como *Sessão por Operação*, *Transação por Operação*, ou ainda *Conexão por Operação* (*Session/Connection per Operation*).

Seria possível agrupar as operações que precisam da mesma conexão/transação dentro de um mesmo método do DAO:

```
public void salva(Projeto projeto, Gerente gerente) {
    Connection connection = new ConnectionFactory().getConnection();
    // ...

    connection.close();
}
```

Mas, neste caso, o DAO acabaria com uma grande quantidade de métodos, a maioria formada praticamente pela combinação de outros. Este é um claro exemplo de péssima divisão de responsabilidades, já que o DAO trabalha demais, abre conexão, cuida de transação, fecha conexão, além da tarefa que realmente lhe cabe: acessar dados. Qualquer novo tipo de transação da aplicação exige um novo método no DAO.

De alguma forma é preciso permitir que a mesma conexão seja utilizada em diversos métodos do DAO, guardando-a em um atributo. Precisamos também de um local para abrir a conexão:

```
public class ProjetoDao {
    private Connection connection;

    public void inicia() {
        this.connection = new ConnectionFactory().getConnection();
    }
}
```

Criar um método para iniciar o DAO e abrir a conexão resolve um problema, mas gera outro. Podemos nos esquecer de invocar o método `inicia`, fazendo com que `Connection` seja `null`. O objeto DAO não é um **Bom Cidadão** (*Good Citizen Pattern*),[145] já que pode ser instanciado sem que todas as suas dependências estejam resolvidas.

Para que um objeto seja um bom cidadão, ele deve sempre estar em um estado consistente. Para tanto, o uso do construtor é essencial, no qual são preenchidas todas as dependências necessárias para que o objeto possa trabalhar adequadamente, sem a subsequente necessidade de invocar *setters* ou outros métodos de inicialização e configuração. O princípio ainda vai além, definindo algumas boas práticas na manipulação de exceções, *logging* e programação defensiva, tornando esses objetos mais seguros em relação a código de terceiros.

A seguinte versão do DAO usa o construtor para resolver o problema e força a aquisição da conexão no momento de sua instanciação:

```
public class ProjetoDao {
    private final Connection connection;

    public ProjetoDao() {
        this.connection = new ConnectionFactory().getConnection();
        // ::poderia também abrir uma transação::
    }

    public void fecha() {
        // ::poderíamos consolidar a transação::
        this.connection.close();
    }
}
```

A conexão é aberta no construtor, porém, Java não tem destrutores. Mesmo que tivesse, ou o programador usasse o método `finalize`, não seria uma solução adequada, já que não há a garantia de quando um objeto será coletado pelo *Garbage*

Collector. É fundamental ter controle sobre onde e quando as conexões são abertas e fechadas, tendo em vista que este é um recurso caro, tal como descritores de arquivos, threads, sockets e outros que usam de I/O ou outras chamadas ao sistema operacional.

Como o nosso DAO já detém a responsabilidade de criar a conexão, é seu papel fechá-la. O método `fecha` parece ser a solução, já que agora é possível instanciar o DAO e invocar várias operações dentro da mesma conexão/transação. Ainda assim, esta solução possui alguns problemas:

- O método `fecha` quebra o encapsulamento, já que expõe um detalhe específico desta implementação de DAO que estamos usando; existe uma conexão que deve ser fechada. Se a implementação do DAO for trocada por uma que persista em arquivos XML, por exemplo, não haveria mais necessidade de ter o método `fecha`, pois não existiria mais nenhuma conexão a ser fechada. Ao mesmo tempo, não se pode simplesmente remover o método `fecha`, já que ele está na interface pública dos DAOs; isso quebraria todos os seus clientes. É nesse momento que começam a surgir as implementações vazias de métodos.

- A boa prática diz: "*se abrir, feche*"! Desta forma, evita-se espalhar a responsabilidade de gerenciar o recurso que estamos manipulando. O problema é que o DAO abre a conexão, mas não sabe quando fechá-la, e delega esta tarefa a quem o utiliza. Os usuários da classe ficam responsáveis por invocar o método `fecha` e saber onde fechar algo que nem foram eles que abriram. A responsabilidade de gerenciar conexões fica assim muito espalhada pelo sistema; todos são responsáveis por saber que uma conexão existe, quando é aberta e quando deve ser fechada.

- Não há erros de compilação por não invocar o método `fecha`, portanto, ninguém é obrigado a invocá-lo. Como a responsabilidade se espalha por todo o sistema, a chance de se esquecer de fechar a conexão aumenta muito, para não dizer que é inevitável. Isso gera problemas de vazamento de conexões, que não são fechadas e podem resultar no desastroso efeito do banco rejeitando novas conexões quando já existirem muitas abertas.

- Fechar a conexão costuma ser mais complexo que uma simples invocação ao método `close`. Pode haver necessidade de controlar as exceções ou executar *rollback* no caso de uma possível transação deixada aberta. Se houver um pool, fechar pode significar devolver a conexão para lá. Para não repetir esse código

em todos os DAOs, poderíamos encapsulá-lo, tal como fizemos com a criação na `ConnectionFactory`. Mas ainda precisamos nos lembrar de chamar esse novo método em todos os lugares, espalhando novamente a responsabilidade.

- Para piorar, nesta abordagem ainda não é possível salvar um `Projeto`, atualizar um `Usuario` e remover uma `Historia` usando a mesma conexão, ou dentro da mesma transação. Como cada DAO possui sua própria conexão, não há como compartilhar a mesma entre vários deles; o escopo de existência de uma conexão é interno ao de um DAO.

A solução para todos esses problemas é tirar do DAO a responsabilidade de gerenciar abertura e fechamento de conexões. Sua única responsabilidade (*Single Responsibility Principle*)[120] será tratar das operações de manipulação do banco, mas sem a preocupação de gerenciar conexões. Aumentamos sua coesão e diminuímos seu acoplamento com outras partes do sistema.

Mas não podemos simplesmente remover todo o código relacionado a `Connection`. Para executar seus comandos SQL, ele ainda precisa de uma conexão. Em vez, então, de criar uma nova conexão no DAO, ele passa a receber uma conexão já criada por alguém. E, com esta conexão, poderá executar sua responsabilidade de manipular o banco de dados sem outras preocupações. Receber a conexão em vez de criá-la é um exemplo do princípio da **Injeção de dependências** (*Dependency Injection – DI)*. Uma forma fácil de codificar seria recebendo no construtor da classe:

```
public class ProjetoDao {
    private final Connection connection;

    public ProjetoDao(Connection connection) {
        this.connection = connection;
    }

    public void adiciona(Projeto p) {
        // ::usa a connection::
    }

    // ::não precisamos de um método fecha()::
}
```

Ao dizer que os objetos não vão mais atrás de suas dependências, mas que agora devem apenas recebê-las de alguém, invertemos quem está no controle de gerenciar

a dependência. Em vez de fazer os objetos irem atrás daquilo que precisam, o programador faz com que eles recebam os objetos já inicializados e preparados para uso. É o que se chama de **Inversão de Controle** (*Inversion of Control – IoC*). Injeção de dependências é uma das formas mais comuns de IoC. [119],[89]

Este conceito teve origem há um bom tempo, no paper de Richard Sweet em 1988,[193] que introduziu o **Hollywood Principle**: "*Don't call, us we will call you*!" (não nos chame, nós o chamaremos!). O objetivo é, ao invés de decidir o momento certo de invocar cada componente, deixar-se ser invocado quando adequado. Ao invés de decidir o melhor momento para invocar certo componente, deixa-se ser invocado por alguém que vai julgar o momento correto. Em DI, vemos esta prática ao deixar de invocar o código de gerenciamento da conexão e passar a ser invocado, por exemplo, no construtor, com tudo já pronto para uso Figura 4.1.
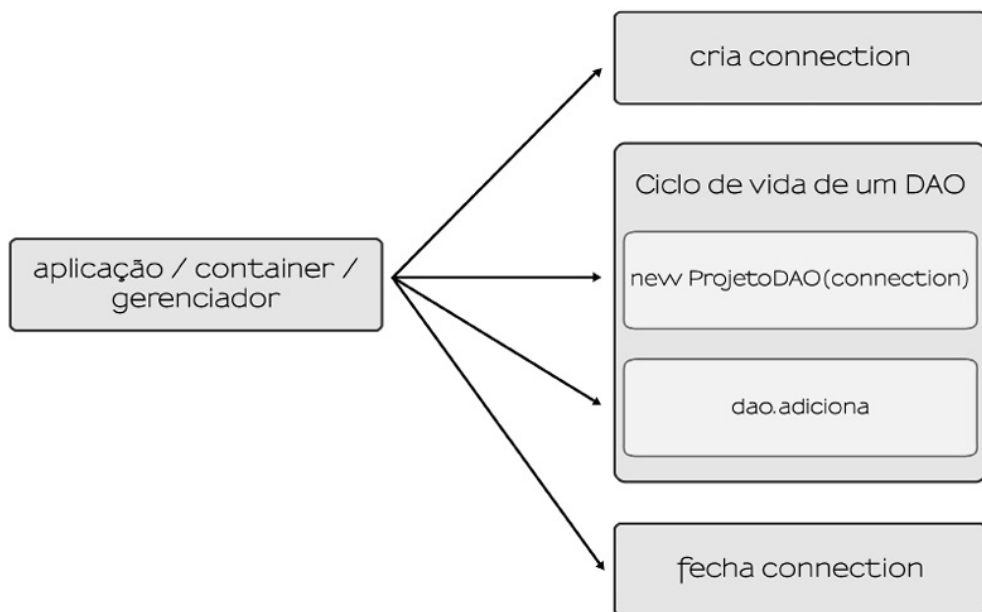
Figura 4.1: Não nos chame, nós o chamaremos: Hollywood Principle

Mas a inversão pode acontecer em outros cenários, que não no gerenciamento de dependências. Uma aplicação visual pode precisar saber quando certo botão é pressionado. Em vez de verificar o estado do botão de tempos em tempos, pode-se apenas registrar um listener que será notificado quando o evento ocorrer. O código

de tratamento do evento passa a ser invocado pelo botão, em vez de invocar métodos para descobrir seu estado. O controle foi invertido. Outro exemplo são os containers Web e EJB do Java EE, que invocam no momento certo os métodos de uma Servlet ou os callbacks de um EJB. **DI é uma das formas de implementar IoC, mas não a única.**

Voltando ao exemplo de DI, resta a questão de quem gerenciará a conexão. Aquele que deseja usar o DAO, precisará se preocupar com isso. Imagine um sistema Web com centenas de Actions diferentes, todas envolvendo o uso de algum DAO. Antes, podia-se simplesmente criá-lo sem se preocupar com conexão; agora passa-se a se preocupar com a conexão:

```
public class AdicionaProjetoAction {
    public void execute() {
        Connection connection;
        connection = new ConnectionFactory().getConnection();
        ProjetoDao dao = new ProjetoDao(connection);
        dao.adiciona(...);
    }
}
```

Usar a `ConnectionFactory` alivia um pouco do problema, mas não é uma boa solução. A action, além de ter responsabilidades normais ligadas a Web, como manipular *request* e *response*, agora também está acoplada a `Connection`, `ConnectionFactory` e DAO. Como diminuir esse acoplamento?

Injeção de dependências novamente. A única responsabilidade da action é adicionar um projeto no DAO, e, para isso, só necessita de um DAO, não precisa se preocupar com os detalhes de sua criação ou de abertura de conexão:

```
public class AdicionaProjetoAction {
    private final ProjetoDao dao;

    public AdicionaProjetoAction(ProjetoDao dao) {
        this.dao = dao;
    }

    public void execute() {
        this.dao.adiciona(...);
    }
}
```

Pode parecer que estamos apenas empurrando o problema, já que alguém terá que criar essa Action. E, agora, instanciá-la exige a existência de um DAO, que, por sua vez, exigirá uma conexão. O problema parece aumentar. Mas o segredo de DI é pensar em cada componente isoladamente, de uma maneira quase egoísta com relação ao restante do sistema. É justamente o princípio de Hollywood na prática, pensando em cada classe como o ponto mais importante da aplicação e empurrando responsabilidades secundárias para outra classe lidar.

Mas, claro, alguém terá que resolver esse emaranhado de dependências em alguma hora. Será algum componente do sistema cuja única responsabilidade seja exatamente a ligação (*wiring*, ou amarrar, injetar) das dependências. É o componente que estará no controle da aplicação, para que todo o resto possa usar DI e IoC. Na injeção de dependências, existe um dependente (o DAO) e uma dependência (a conexão); e quem amarra tudo isso é um provedor (*provider*). Por ser algo tão importante e bastante comum em diversas aplicações, normalmente usa-se um framework pronto como provider. Spring, Pico Container, Google Guice e o CDI do Java EE 6 são exemplos de container de DI, como veremos no tópico seguinte.

É importante ressaltar, porém, a diferença entre a prática de Injeção de Dependências e o uso de algum framework específico. Inverter o controle e declarar suas dependências para ser injetadas por alguém é considerado uma boa prática de design, e é até mesmo um design pattern bem aceito.[151] Devemos programar nossas classes com DI sempre que possível.

Mas isso não implica que sejamos obrigados a usar algum framework como os citados anteriormente. Há quem defenda evitar o uso de frameworks de DI e que o próprio código de gerenciamento das dependências seja implementado, já que é algo que costuma ser fácil de ser escrito pelo programador, além de ficar mais explícito como os objetos se relacionam.[118],[163] É uma opinião que gera bastante polêmica, já que os frameworks modernos são simples de usar e livram o programador de mais uma dor de cabeça, por menor que ela possa ser. Há ainda uma discussão: se, em linguagens mais flexíveis que Java, como Scala ou Ruby, a própria linguagem já não traz recursos equivalentes aos frameworks de DI.[20],[107]

Independentemente da forma de se implementar, é praticamente consenso que o uso de Injeção de Dependências é uma prática de design quase obrigatória. Com as vantagens de diminuir o acoplamento entre os componentes, aumentar a coesão das partes do sistema e promover uma melhor separação de responsabilidades, o uso de DI é altamente recomendado.

4.3 Considere usar um framework de Injeção de Dependências

Discutimos no tópico anterior a importância do uso da inversão de controle e da injeção de dependências. Mesmo sendo considerada uma boa prática de design, ainda traz alguns desafios, pois gerenciar as dependências de todas as classes do projeto não é tarefa fácil. Há muitas complicações envolvidas, como tratar das dependências de dependências, cuidar da criação e liberação dos componentes e gerenciar o momento em que tudo isso deve ser feito.

Visando facilitar esse trabalho, surgiram diversos frameworks de injeção de dependências. Um dos principais, em importância e em uso, é o **Spring Framework**, idealizado por Rod Johnson. Pioneiro em DI, o Spring nasceu com suporte apenas à chamada injeção via setter:

```
public class AdicionaProjetoAction {
    private ProjetoDao dao;

    public setDao(ProjetoDao dao) {
        this.dao = dao;
    }

    // ...
}
```

Sua configuração, durante muito tempo, suportava apenas XML, como no trecho a seguir:

```
<beans ...>
    <bean id="adicionaAction’’ class=’’br.com.arquiteturajava.AdicionaProjetoActio
        <property name="dao’’>
        <ref local="projetoDao’’/>
        </property>
    </bean>
</beans>
```

Essa antiga abordagem de usar setters para injeção, porém, é bastante problemática. Se a dependência, por algum motivo, não estiver disponível, a classe será criada em um estado inconsistente, não será um *Good Citizen*. Invocar os métodos de negócio pode causar `NullPointerExceptions` indesejadas, já que a referência não terá

sido preenchida. Além disso, sem a utilização de uma ferramenta de injeção de dependências, o desenvolvedor é obrigado a lembrar de invocar os setters apropriados antes de efetivamente usar o objeto, como ao escrever testes de unidades.

Com o tempo, o Spring passou a suportar injeção via construtores, além de configurações com anotações, diversas convenções e simplificações.

O framework cresceu a ponto de integrar também diversos outros serviços e funcionalidades, como AOP, persistência, controle transacional, MVC e, mais recentemente, até uso com outras linguagens e dentro da plataforma de cloud da VMWare. O componente central em tudo isso sempre foi o container de injeção de dependências, que consagrou o Spring.

Mas foi o **PicoContainer**, outro dos principais frameworks de DI, o primeiro a implementar injeção via construtores. Esse tipo de injeção soluciona o problema do método setter ao exigir que as dependências necessárias para o funcionamento do objeto sejam passadas ao instanciá-lo. Por este motivo e, por respeitar o princípio do bom cidadão, o uso de injeção por construtor é o mais utilizado atualmente.

Idealizado por Paul Hammant e Aslak Hellesoy, o Pico é um framework que carrega até no nome a ideia de minimalismo e simplicidade. Seu único foco é DI, sem possuir todos os desdobramentos que o Spring e outros frameworks possuem. Ele surgiu questionando o excesso de configurações em XML do Spring e outros frameworks, e trouxe uma abordagem de configuração programática através da própria linguagem Java, com as vantagens de evitar erros em tempo de compilação e autocompletar de IDEs.

```
// ::exemplo de classe com dependência no construtor::
public class AdicionaProjetoAction {
    private final ProjetoDao dao;

    public AdicionaProjetoAction(ProjetoDao dao) {
        this.dao = dao;
    }

    public void execute() {
        this.dao.adiciona(...);
    }
}

// ::configuração programática em algum outro lugar::
MutablePicoContainer pico = new DefaultPicoContainer();
```

```
pico.addComponent(ProjetoDao.class);
pico.addComponent(AdicionaProjetoAction.class);
```

Mesmo mudando a forma, a configuração do Pico ainda seria externa, e as classes do desenvolvedor não teriam ligação forte com o framework. É até possível migrar de um container para o outro apenas portando a configuração, sem mexer nas classes da aplicação. Hoje, o PicoContainer suporta também configuração por anotações, injeção por setters e outros recursos mais complexos para gerenciamento das dependências.[157]

Outra forma de injeção, além dos setters e construtores, que aparece bastante no EJB 3.x mas também em outros frameworks, é a injeção diretamente em atributos. O apelo desta abordagem é a simplicidade do código da classe, que não precisa nem de setter nem de construtor, mas não costuma ser a abordagem preferida para uso de DI. Escrever um teste de unidade para uma classe que recebe as dependências vai exigir o uso de reflection para conseguir injetá-la. Para evitar isto, os frameworks que optam por este tipo de injeção criaram outros frameworks, que servem simplesmente para facilitar a escrita desses testes. Isso mostra o quanto o design fica acoplado com a ferramenta específica, mas este tipo de acoplamento deve ser evitado ao máximo.

Com o crescimento do uso de anotações em Java, os frameworks de DI passaram a oferecer seu uso como alternativa para a configuração. O **Google Guice** foi um dos primeiros a oferecer configuração dos componentes e das injeções usando anotações. Com a anotação `com.google.inject.Inject` fazemos a única configuração necessária:

```
public class AdicionaProjetoAction {
    private final ProjetoDao dao;

    @Inject
    public AdicionaProjetoAction(ProjetoDao dao) {
        this.dao = dao;
    }

    public void execute() {
        this.dao.adiciona(...);
    }
}
```

O uso de anotações simplificou as configurações, mas trouxe à tona a discussão sobre acoplamento com o framework. No Spring, por exemplo, é necessário anotar

os pontos de injeção com `@AutoWired`, ou no Guice, com `@Inject` – ambas específicas de cada um deles. Trocar de framework nesse cenário é bastante trabalhoso; o Pico, por exemplo, até tenta minimizar o problema suportando anotações de outros frameworks, como o Guice, mas esse não é o melhor cenário.

A possível solução surgiu com uma proposta do Guice e do Spring para padronização de um conjunto simples de anotações. A JSR-330 especificou um novo pacote `javax.inject` com poucas anotações comuns a serem usadas para configurar as classes do usuário, como a `@javax.inject.Inject` para uso nos pontos de injeção espalhados pelo código. Essa JSR não especifica, porém, como funciona o framework, suas configurações e o processo de DI em si, mas apenas a interface entre o código da aplicação e o framework. Foi um passo importantíssimo para diminuir o acoplamento com frameworks específicos. Embora ainda haja o uso de configuração no código, são anotações genéricas, portáveis e especificadas.

Mas o grande passo para a plataforma Java, além da JSR-330, foi o lançamento do **CDI – Context and Dependency Injection**, a JSR-299, junto com o Java EE 6. Entre as várias funcionalidades dessa especificação, muitas inspiradas pelo JBoss Seam, está uma ferramenta completa de injeção de dependências para a plataforma Java EE. Anteriormente, no EJB 3, algumas poucas funcionalidades de injeção foram adicionadas, de maneira muito limitada: apenas alguns tipos eram disponíveis e limitados à injeção por setter ou atributo. O CDI traz uma especificação à altura dos frameworks mais consagrados, integrada à JSR-330 e disponível em toda stack do Java EE 6.

O mesmo código do exemplo de Google Guice funcionará com CDI, apenas trocando o pacote da anotação para `javax.inject`. Nenhum código Java é necessário, e uma simples anotação traz DI para beans do JSF, EJBs e até Servlets e outros frameworks. Outro exemplo que mostra a simplicidade do CDI é injetar um `PersistenceContext` da JPA dentro do DAO, usando factories produtoras de beans com a simples anotação `@Produces`:

```
// o DAO com a configuração do ponto de injeção
public class ProjetoDao {
    private final EntityManager manager;

    @Inject
    public ProjetoDao(EntityManager manager) {
        this.manager = manager;
    }

    // ... restante do DAO ...
```

```
}

// a classe que produz o bean
public class JPAUtil {

    public @Produces EntityManager criaEntityManager() {
        // implementação que devolve um novo EntityManager customizado
    }
}
```

Mas é possível atender a cenários ainda mais complexos com pouco esforço. Outro exemplo seria criar diferentes EntityManagers para diferentes DAOs, suportando um caso de uso em que lógica da diretoria tem acesso a um banco de dados restrito. Podemos escrever dois métodos produtores e diferenciar o bean gerado com um `Qualifier`:

```
// uma anotação de negócio para diferenciar os beans
@Qualifier
@Retention(RetentionPolicy.RUNTIME)
public @interface Diretoria { }

// ::2 produtores, um implicitamente @Default e outro para diretoria::
public class JPAUtil {
    public @Produces EntityManager criaEntityManager() {
        // ::implementação que devolve um novo EntityManager customizado::
    }

    public @Produces @Diretoria EntityManager criaEMDiretoria() {
        // ::implementação que devolve novo EntityManager com superpoderes::
    }
}

    // ::DAOs normais continuam recebendo EntityManager normal::
public class ProjetoDao {
    private final EntityManager manager;

    @Inject
    public ProjetoDao(EntityManager manager) {
        this.manager = manager;
    }
}
```

```
// a Diretoria recebe o DAO especial
public class RelatoriosSigilososDao {
    private final EntityManager manager;

    @Inject
    public RelatoriosSigilososDao(@Diretoria EntityManager manager) {
        this.manager = manager;
    }
}
```

Há ainda muitos outros recursos interessantes, como suporte a *decorators*, interceptadores de métodos, *listeners* de eventos, e até a possibilidade de usar versões alternativas dos beans, dependendo se o ambiente é de desenvolvimento, testes ou produção. Vários desses recursos estão presentes também em outros frameworks DI, mas o CDI tem ganhado muitos adeptos por ser uma solução oficial e portável. E, além de ser uma especificação, é extremamente fácil, produtivo e com código bastante limpo, muitas vezes mais simples até que outros frameworks.

Outros recursos de containers DI

Quando se pensa em DI, imediatamente se lembra de criação e injeção de objetos. Realmente é este o foco principal, mas retirar de dentro das classes a responsabilidade de criação das dependências abre caminho para diversos outros recursos interessantes.

A primeira questão a se pensar é sobre quem será responsável por finalizar os objetos. Muitos deles não precisam ser somente criados, mas também finalizados, como ao chamar o método `close` de uma `Connection`. Mas de quem é esta responsabilidade agora? Quem criou o objeto costuma ser o responsável por sua finalização. Antes, nosso DAO abria a `Connection` e tinha que fechá-la. Agora, ele apenas recebe uma criada pelo container. Passa a ser responsabilidade deste, portanto, lidar com a finalização e liberação das dependências.

Todos os frameworks citados vêm com recursos importantes para administrar o **ciclo de vida** completo dos objetos gerenciados. Não apenas criar e injetar as dependências, como também saber como finalizá-las. No CDI, por exemplo, a anotação `@Disposes` permite indicar blocos de código para finalização dos componentes. Mas surge uma outra questão: qual é o momento certo de finalizar determinado objeto? E, mesmo com relação à criação, quando instanciar as dependências? E quantos

objetos criar? Deve haver uma única `Connection` na aplicação toda a ser injetada em todos os DAOs? Ou deve ser aberta uma `Connection` cada vez que alguém pedir uma? Ou uma `Connection` para cada usuário, independentemente de quantos DAOs ele use?

Quando se fala de ciclo de vida, é importante saber determinar a forma de gerenciamento de cada objeto, quando e como deve ser criado, e quando deve ser destruído. Estamos falando em definir um **escopo** para a existência de cada objeto. Todos os containers DI modernos são contextuais e permitem a configuração dos escopos. Em uma aplicação Web, em geral, usamos escopos associados a cada request, à sessão do usuário, ou escopo global da aplicação. No XML do Spring, por exemplo, usamos `scope="request"` para configurar que um certo bean deve ser criado a cada request e destruído no final da requisição. Ou, no CDI, podemos usar a anotação `@SessionScoped` para indicar que certo componente está associado à sessão do usuário, e será destruído quando a sessão for invalidada.

Há outros escopos, como o `prototype`, do Spring, que criam uma instância nova a cada ponto de injeção, ou o escopo conversacional do JBoss Seam e do CDI, que permite a um objeto ser usado durante uma sequência de requisições como um wizard, uma conversa. Ou, ainda, no JSF que encontramos o `ViewScope`, além da capacidade extremamente flexível do CDI para se definir novos escopos para a aplicação.

Mas os recursos não param por aí. Com o gerenciamento completo do ciclo de vida dos objetos por parte dos containers, estes disponibilizam ainda mais eventos que somente criação e destruição. É possível configurar callbacks, como o `@PostConstruct` e o `@PreDestroy`, ou, ainda, interceptar execuções dos métodos de negócio dos beans.

Além disso, por ser um objeto criado pelo container e injetado nas classes sem elas saberem exatamente como foi feita a instância, muitos frameworks adicionam recursos especiais aos objetos. É fácil criar *decorators* ao redor do objeto injetado. Ou, ainda, injetar um *proxy* para o objeto real, que pode adicionar recursos como a AOP do Spring e do Guice, ou a poderosa injeção de objetos de menor escopo encontrada no CDI. Tudo isso sem que a classe que recebe a dependência precise conhecer nada a mais sobre o objeto injetado, com acoplamento mínimo e excelente extensibilidade.

E isso para ficar apenas ao redor do serviço de gerenciamento de dependências. Na prática, muitos outros serviços são oferecidos em conjunto com o framework, como no Spring. Embora, como vimos, DI seja uma importante prática de design e podemos usá-la até com gerenciamento manual de dependências, o uso de um

framework rico para fazer este gerenciamento automático é muito recomendado. Com recursos avançados já prontos e de uso cada vez mais simplificado, usar um container DI é algo a ser fortemente considerado.

4.4 Fábricas e o mito do baixo acoplamento

Uma das técnicas de orientação a objetos que mais favorecem o baixo acoplamento é o polimorfismo. Poder criar uma interface comum para diversas implementações possíveis e escolher a implementação mais apropriada para cada situação são recursos bastante poderosos. É como ter um sistema genérico de pagamentos independente da empresa de pagamentos:

```
ServicoPagamentos pgto = new PagamentoViaCielo();

// ::ou:::

ServicoPagamentos pgto = new PagamentoViaRedecard();
```

Ao usar uma interface comum para ambos os serviços, podemos nos acoplar somente à interface e trocar a implementação sem maiores esforços. Independente de qual implementação seja usada, teremos o mesmo código de uso:

```
pgto.setValor(99.90);
pgto.setCartao("4111 1234 5678 0987");
pgto.efetuaCobranca();
```

É excelente podermos trocar de implementação apenas mudando um pequeno pedaço do programa, onde damos o `new`. Mas e se este serviço for usado em vários lugares? Ou, mesmo que seja em apenas um lugar, será responsabilidade deste saber qual empresa de pagamentos deve ser usada? Em um exemplo de loja virtual, poderíamos ter um componente do sistema responsável por finalizar as compras dos usuários. Ele deve receber o pedido, disparar uma ordem, destinada ao serviço de estoque e entregas, e efetuar o pagamento:

```
public class FinalizaCompra {
    public void finaliza(Pedido pedido) {
        ServicoPagamentos pgto = new PagamentoViaCielo();
        pgto.setValor(pedido.getTotal());
        pgto.setCartao(pedido.getCliente().getCartao();
        pgto.efetuaCobranca();
```

```
        // ... ::outras lógicas::
    }
}
```

Parece um código bastante simples, mas mostra um dos maiores pontos fracos do `new`: instanciar diretamente um objeto gera um acoplamento com a implementação usada. Por mais óbvio que isso possa parecer, há graves consequências:

- A classe `FinalizaCompra` passa a estar acoplada com a implementação específica `PagamentoViaCielo`. Se esta classe mudar, por exemplo, para receber um argumento especial com um código da Cielo, ela quebra.
- A classe fica com a responsabilidade de decidir qual implementação deve ser usada para o serviço de pagamentos. É uma grande responsabilidade, algo que tira o foco da responsabilidade primária que seria a simples finalização da compra.
- Se houver mais de uma classe no sistema que precise usar um sistema de pagamentos, todas terão que tomar a decisão de qual implementação usar. Pode haver inconsistências e, mesmo que não haja, a responsabilidade estaria espalhada por diversos lugares.
- A criação do objeto pode ser potencialmente complexa. Talvez seja necessário passar argumentos complicados, e até construir objetos auxiliares para passar para nosso serviço de pagamento escolhido. Deixar toda essa responsabilidade na mão da `FinalizaCompra` não é uma boa ideia.

A solução para encapsular a responsabilidade de criação do objeto específico é dar origem a uma *factory*. Esse importante e famoso design pattern, ao contrário do simples construtor, permite que seja encapsulado o tipo real do objeto instanciado e, mais, centraliza a decisão de criação para que mudanças futuras sejam muito simples. Além disso, se a criação do objeto exigir parâmetros ou uma inicialização complexa, a factory torna tudo mais fácil encapsulando a complexidade do usuário.

Usar uma factory é muito fácil:

```
public class FinalizaCompra {
    public void finaliza(Pedido pedido) {
    ServicoPagamentos pgto = Pagamentos.criaServicoPagamentos();
    // ... ::restante da lógica::
```

```
    }
}
```

A diminuição do acoplamento e o encapsulamento são visíveis. Há diversas opções para criar uma factory: a mais simples é um método estático, como no exemplo, mas há variações, como um objeto de fábrica ou a *Abstract Factory* completa. Mas, em todos os casos, as fábricas, embora melhores que o simples `new`, ainda trazem diversos pontos negativos:

- O ponto mais direto é que agora há um acoplamento com a fábrica em si. Embora seja menos pior que acoplar à implementação do componente, estamos acoplados à fábrica, e qualquer mudança nela pode quebrar nosso código. O método de fábrica, usualmente estático, é um ponto de acesso global ao objeto, e estamos amarrando o projeto todo a este ponto.

- Se existirem vários lugares na aplicação que precisam de um serviço de pagamentos, todos necessitarão chamar essa fábrica. Estamos espalhando a responsabilidade de criação deste objeto por vários lugares diferentes.

- Não é possível usar implementações diferentes em diferentes pontos do programa. Se cada módulo do sistema precisar de um diferente serviço de pagamento, precisaríamos, de alguma forma, indicar para a fábrica qual objeto deve ser criado. Passar um argumento à fábrica, indicando se é Cielo ou Redecard, costuma ser a solução, mas volta o problema de deixar para a classe `FinalizaCompra` a difícil responsabilidade de escolher a implementação.

- Será que o método `finaliza` é o melhor lugar para se criar o serviço de pagamentos? E se precisarmos compartilhar a mesma instância com mais de uma classe? E se a regra de negócio exigir que só podemos criar um serviço de pagamentos por usuário? Como decidir o momento certo de criar o objeto chamando a fábrica? E se o objeto exigir uma finalização, como fechar uma conexão; de quem é esta responsabilidade? É o problema de definir o escopo dos objetos.

- A classe `FinalizaCompra` pode precisar de mais componentes além do `ServicoPagamentos`, como `ServicoEstoque` e `ServicoEntrega`. A solução seria usar fábricas específicas para cada componente, mas isso aumenta ainda mais as responsabilidades da nossa classe, além de, agora, gerar acoplamento com várias outras classes de fábrica.

Alguns dos pontos negativos da factory são sanados pelo uso de um *Service Locator* ou *Registry*. A ideia é diminuir o problema do acoplamento com diversas fábricas diferentes, e de se conhecer pontos de acesso global distintos para cada dependência. Teríamos um único componente global no sistema, cuja responsabilidade é centralizar a localização de todas as dependências possíveis e prover um ponto de acesso único e simples para quaisquer componentes que forem necessários em qualquer parte do sistema.

Uma das formas mais comuns de se criar um service locator é prover um único método que recebe como argumento qual componente se deseja obter. Isto pode ser feito através de um nome em String (como em JNDI, por exemplo), ou até com soluções mais complexas, passando a interface desejada:

```
public class FinalizaCompra {
    public void finaliza(Pedido pedido) {
        // usando nomes para os componentes
        ServicoPagamentos pgto = Locator.get("servico-pagamentos");
        // usando a interface
        ServicoPagamentos pgto = Locator.get(ServicoPagamentos.class);
    }
}
```

Com o service locator global, não importa quantas dependências a classe precisa, é possível obter todas a partir dele. Ainda temos um acoplamento ao service locator, mas é algo bem mais leve que nas fábricas, já que será um único ponto de acesso global para todo o sistema. Usando a versão com nomes para os componentes, conseguimos ainda resolver o problema de usar diferentes implementações em diferentes partes do sistema, bastando dar nomes diferen-tes para as dependências.

Com algum esforço razoável, é possível ainda que o service locator resolva o problema do escopo de criação das dependências. Seria possível que ele tivesse um cache interno para guardar dependências, que devem ser instanciadas uma única vez em toda a aplicação e compartilhadas com todos. Com um pouco de mágica e possivelmente `ThreadLocal`, seria possível até que houvesse um pequeno cache para permitir compartilhar o mesmo objeto dentro apenas do mesmo request.

Mas essa solução ainda deixa alguns pontos em aberto. Por mais que tenhamos diminuído o acoplamento com coisas externas, ainda temos o acoplamento com o service locator. Pior ainda, *todas* as classes do sistema estarão acopladas a ele, e qualquer mudança simples passa a ser extremamente traumática para todo o sistema. Há ainda o problema do ponto de acesso global. As classes precisam conhecer esse ponto

de acesso e ainda saber a localização exata de suas dependências, por mais encapsulado que isso esteja dentro do service locator.

Porém, o problema mais grave é que **Service Locator não é Inversão de Controle**. Como a aplicação precisa invocar diretamente o Service Locator para obter sua dependência, a aplicação tem o controle. Injeção de dependências resolve os últimos problemas ao inverter o controle do código de resolução de dependência, tirando das classes da aplicação a responsabilidade de resolver suas próprias dependências.

Martin Fowler diz[65] que a grande decisão é se queremos estar acoplados ao Service Locator em toda a aplicação ou não. A Inversão de Controle remove esse acoplamento, mas pode tornar mais difícil enxergar o que a aplicação faz exatamente, e até seu fluxo de execução. Por definição, fica difícil enxergar quem está no controle, e isto pode eventualmente ser um problema.

No geral, as vantagens da inversão de controle superam seus pontos negativos, por isso seu uso é tão recomendado. Com Injeção de Dependências, não há o acoplamento a um ponto de acesso global. Não há necessidade de as classes conhecerem a localização de suas dependências.

Usando DI, as classes possuem menos responsabilidades e preocupações, já que alguém estará no controle do gerenciamento das dependências. Como a injeção é feita por alguém externo, sua configuração é externa à classe, e decidir qual implementação deve ser injetada em qual lugar não é mais sua responsabilidade. Além disso, como vimos, as ferramentas modernas de injeção de dependências conseguem ainda trazer outras vantagens, como controle completo de ciclo de vida das dependências, incluindo suas finalizações, escopos bem definidos, entre outras.

Singletons, Factories e os containers de DI

Singleton é um dos design patterns com pior fama. Chega a ser chamado por muitos de *antipattern*. Em 2009, durante uma entrevista em comemoração aos 15 anos do livro *Design Patterns*, ao ser perguntado sobre quais mudanças faria no livro em uma hipotética revisão, Erich Gamma afirmou que tiraria Singleton dele. Mais, o autor ainda afirmou que singletons seriam um *design smell*.[151]

O singleton, à primeira vista, parece bastante simples e inofensivo. Com frequência, há a necessidade de se modelar na aplicação a restrição de instância única para certas classes. O singleton aparece como uma solução elegante para impedir a criação indiscriminada de instâncias, controlando o acesso à instância única criada internamente. Um nome bastante comum para o método que expõe a instância única é `getInstance`:

```
public class Configuracoes {
    private static final Configuracoes INSTANCE = new Configuracoes();
    private Configuracoes(){}

    public static Configuracoes getInstance() {
        return INSTANCE;
    }

    public String get(String chave) {
        // ... ::implementação::
    }
}
```

Este exemplo representa uma classe que guarda configurações globais para o sistema. É implementado com um código simples, que cria a instância no atributo e disponibiliza um ponto de acesso através do método `getInstance`. Há outras possíveis formas de implementar, como, por exemplo, a criação *lazy* da instância, ou ainda usando uma `enum` do Java 5, como o livro *Effective Java* propõe.

Seja qual for a forma escolhida, a primeira pergunta a se colocar é de cunho bem prático e técnico: singletons realmente existem? A verdade é que a restrição da instância única pode ser facilmente burlada usando classloaders diferentes. Ou, ainda, forçando a criação de novas instâncias com reflection. Em um ambiente cluster, então, a questão é quase filosófica. Como encarar os singletons se há diferentes instâncias da aplicação rodando em diferentes máquinas?

Mas, deixando de lado as questões técnicas de como burlar um singleton, o grande problema desse pattern é de design. As classes que forem depender do singleton precisarão obter a instância única a partir do seu método `getInstance`, um ponto de acesso global que gera um acoplamento de todos os usuários com a localização do objeto:

```
public class AdicionaAction {
    public void execute() {
        Configuracoes config = Configuracoes.getInstance();

        String dado = config.get("informacoes");
        // ...
    }
}
```

De certa forma, o método `getInstance` pode ser encarado como uma fábrica que sempre produz a mesma coisa, ou um Registry de um único objeto. Há os mesmos problemas discutidos anteriormente de Factory e de Service Locator. Há talvez até um acoplamento maior, pois em geral o usuário do singleton acaba sabendo que aquilo é um singleton, ainda mais pela semântica associada a `getInstance`. O usuário acaba carregando o conhecimento de que se trata de um singleton, ou seja, acopla-se com um detalhe interno da criação de instâncias de uma outra classe. No dia em que a classe `Configuracoes` precisar deixar de ser um singleton e passar a ter várias instâncias, todo o restante do sistema será afetado.

Na mesma entrevista em que critica o Singleton, Erich Gamma propõe um novo *design pattern* que poderia ser incluído no livro: a Injeção de Dependências, justamente a solução para os problemas citados.

```
public class AdicionaAction {
    private final Configuracoes config;

    public AdicionaAction(Configuracoes config) {
        this.config = config;
    }

    public void execute() {
        String dado = this.config.get("informacoes");
        // ...
    }
}
```

A action passa a não mais saber que se trata de um singleton. Se um dia este fato mudar e várias instâncias passarem a ser necessárias, a action não sofrerá qualquer alteração. E mais: como teremos alguém no controle das dependências e criação dos objetos, seria possível até que a classe `Configuracoes` não fosse mais codificada como um singleton. Não que a necessidade de ter uma instância única não seja válida, mas esta pode ser a responsabilidade do container que gerencia as dependências, e a própria classe `Configuracoes` poderia ter uma responsabilidade a menos ao não precisar se preocupar em criar e cuidar de sua instância única. Se um dia forem necessárias várias instâncias, nem a classe `Configuracoes` sofreria modificações, mas apenas o componente que controla as dependências saberia deste fato.

Na verdade, todos os containers de DI suportam que se configure certo componente, uma simples classe, como um singleton, ou seja, uma instância única. Mas é

uma configuração interna ao framework. No Spring, por exemplo, é possível configurar que qualquer classe do sistema seja um bean com instância única compartilhada. Basta usar o atributo `scope` no XML, indicando justamente o valor `singleton`:

```
<bean id="configuracoes" class="br.com.arquiteturajava.di.Configuracoes" scope="singleton" />
```

Ou, no PicoContainer, ao adicionar programaticamente um componente, podemos indicar que a instância criada deve ser única e compartilhada por todos:

```
MutablePicoContainer pico = new DefaultPicoContainer();
pico.as(CACHE).addComponent(Configuracoes.class);
```

No Java EE 6, uma das maiores adições ao EJB em sua versão 3.1 foi o *Singleton Bean*. Uma simples anotação `@Singleton` instrui o container que aquele EJB deve ter uma instância única compartilhada em toda a aplicação:

```
@Singleton
public class Configuracoes {
    public String get(String chave) {
        // .. implementação
    }
}
```

Google Guice, CDI e qualquer outro framework de DI possuem recursos parecidos. Na prática, o que observamos é que o **Singleton deixa de ser um design pattern implementado em código e evolui para apenas uma configuração dos frameworks de DI** para instruí-los em como será a política de criação de objetos de certa classe.

E não é só o singleton que encontrou seu espaço dentro dos containers de DI. Factories, de certa forma, também evoluíram para fazer parte do container. Assim como a motivação da instância única dos singletons ainda pode ser válida mesmo ao usar DI, as motivações das fábricas também continuam válidas. Estas têm a vantagem de encapsular muito bem a criação de objetos complexos, que demandem diversos passos, algo que poderia complicar o trabalho de um framework automatizado.

No tópico anterior foi possível observar isso no CDI com sua anotação `@Produces`. Esse recurso nada mais é que uma forma de indicar ao framework que determinado código é um método de fábrica para certo tipo de dependência, e que o container deve invocá-lo para obter novos objetos sempre que necessário:

```
public class JPAUtil {
    public @Produces EntityManager criaEntityManager() {
        // implementação que devolve um novo EntityManager customizado
    }
}
```

Mas repare que uma aplicação jamais invocaria esse método `criaEntityManager` diretamente. Fazê-lo traria de volta os problemas de ponto de acesso global das fábricas e o alto acoplamento. Esse método é uma fábrica que será invocada somente pelo container. No Spring, temos o mesmo efeito usando seus `FactoryBeans` e, no Pico, seus `ComponentAdapter`.

Nos lugares da aplicação onde for necessário um `EntityManager`, basta declarar a dependência e o container cuidará da invocação da fábrica internamente. É a evolução das fábricas com todas as suas vantagens, mas em um contexto de Injeção de Dependências de baixo acoplamento e sem acesso global.

O Service Locator que há dentro de cada container DI

Todo container de DI, internamente, é um grande Service Locator, um imenso Registry para nossos objetos. É possível até encarar o framework desta forma. No Spring, por exemplo, depois de configurados os beans no XML, é possível obter referências para objetos desejados pelo nome configurado:

```
ApplicationContext container;
container = new FileSystemXmlApplicationContext("spring.xml");

AdicionaProjetoAction action;
action = (AdicionaProjetoAction) container.getBean("adicionaAction");

action.execute();
```

O código anterior mostra como o Spring facilita a obtenção do objeto da Action sem se preocupar se ela vai receber um DAO como argumento, ou até se o DAO tem uma conexão como dependência, que, por sua vez, precisa dos dados de conexão, usuário e senha. Todas as dependências são resolvidas, basta indicar o nome do bean no XML (`adicionaAction`) e temos uma instância pronta para uso.

Com o PicoContainer não seria diferente. Após configurar todas as dependências programaticamente, como vimos no tópico anterior, seria possível obter as referências desejadas diretamente no container. A diferença é que passamos o tipo do objeto na hora de obtê-lo:

```
PicoContainer container = ....

AdicionaProjetoAction action;
action = container.getComponent(AdicionaProjetoAction.class);

action.execute();
```

Guice, CDI e outros frameworks têm código semelhante. Só há um problema: estamos usando um grande Service Locator, e não Injeção de Dependências. **Invocar diretamente o container de DI na aplicação não é Inversão de Controle, e é uma má prática**.

Pense na classe que eventualmente teria esse código de invocação direta ao container para obter uma `AdicionaProjetoAction`. Seu objetivo não é manipular o container, mas, sim, obter um objeto de action para invocar seu método `execute`. Se sua responsabilidade é apenas utilizar a action, poderíamos também usar Injeção de Dependências nessa classe e fazê-la receber uma `AdicionaProjetoAction` no construtor, sem precisar de referência para o container nem invocá-lo explicitamente para obter sua dependência.

Mas há uma questão não resolvida. Inversão de Controle não significa falta de controle. Alguém precisa estar no controle. Mesmo usando um framework de DI, alguém precisa iniciá-lo, dar o passo de inicialização e chamar explicitamente o primeiro bean. É preciso um código inicial, que não será Inversão de Controle, para que o restante da aplicação esteja com o controle invertido.

Em uma aplicação gerenciada por um container Web ou EJB, o próprio servidor pode fazer esse passo inicial de criar o container de DI, já que ele está no controle. Uma aplicação CDI que rode em um container Java EE 6, por exemplo, nunca precisará invocar qualquer código do CDI para inicializar o container ou invocar as dependências explicitamente. Com CDI no servidor, basta jogar as classes anotadas corretamente que o CDI fará todo o controle da aplicação automaticamente.

Porém, nem sempre a aplicação roda em um ambiente gerenciado pelo container do servidor. Uma aplicação Desktop no método `main`, por exemplo, teria que inicializar o framework de DI manualmente. Este método é o controle inicial de toda aplicação Desktop, e, se queremos usar Inversão de Controle no restante da aplicação, será seu papel disparar o código mínimo necessário para controlar a aplicação. Podemos dizer que seria necessário escrever um código de *bootstrap* da aplicação que inicia o framework e controla todo o restante do sistema. Esse código, embora importante e necessário, não é Inversão de Controle e, portanto, deve ser o menor

possível, sem lógica de negócios e com a única responsabilidade de controlar a aplicação.

Referenciar o container no meio da aplicação e invocar seus beans diretamente é uma péssima prática, contrária a tudo que se viu de Inversão de Controle e Injeção de Dependências. A única situação justificável é nos raros casos em que precisamos escrever um código curto de *bootstrap* da aplicação.

4.5 Proxies dinâmicas e geração de bytecodes

Os recursos de orientação a objetos e a API básica de reflection podem não ser suficientes para separar códigos de infraestrutura e de domínio de maneira completamente transparente.

Considere a invocação de um método de um objeto que esteja em outra máquina virtual. É comum existir um repositório central de dados, que realiza pesquisa em um estoque de livros, para saber a disponibilidade destes. Como acessar essa informação sem expor o banco de dados?

Se nosso servidor está rodando Java, pode haver um objeto que representa a `Livraria` e possui métodos que precisam ser acessados a partir de uma outra máquina virtual. Uma ideia poderia ser criar uma classe, que controla o acesso aos objetos dessa JVM remota, e executar os métodos que desejarmos:

```
ClasseControladoraDoServidor.executa("livraria", “buscaAutor”);
```

Nessa abordagem, fica difícil passar parâmetros, receber retornos diferentes, além de não checar o correto funcionamento da chamada em tempo de compilação, podendo surgir erros em tempo de execução. Também é muito trabalhoso ser obrigado a cuidar de tanta infraestrutura, desde trabalhar com `Sockets` puras até definir protocolos e gerenciar situações de falha.

Seria mais interessante se a aplicação cliente simplesmente tivesse uma referência a um objeto local, que pudesse ser utilizado para invocar métodos em um objeto que está na aplicação servidora em outra JVM. Algo como `livraria.buscaAutor("turgueniev")`.

Para chegar a uma solução elegante como esta, inicialmente há necessidade de uma classe para poder trabalhar com o identificador do objeto remoto, o nome do método a ser invocado e seus parâmetros. Esse `Controlador` fica responsável por repassar essas informações à máquina virtual remota:

```
public class Controlador {
    public <T> T invoca(Long id, String metodo, Object... parametros)
    throws IOException, ClassNotFoundException {
        Socket socket = new Socket("192.168.87.17", 3040);
        try {
            OutputStream saida = socket.getOutputStream();
            InputStream entrada = socket.getInputStream();
            ObjectOutputStream writer = new ObjectOutputStream(saida);
            writer.writeObject(id);
            writer.writeObject(metodo);
            writer.writeObject(parametros);
            ObjectInputStream reader = new ObjectInputStream(entrada);
            return (T) reader.readObject();
        } finally {
            socket.close();
        }
    }
}
```

Dado que o cliente conhece o objeto referenciado no servidor, ele pode agora executar métodos através dessa classe controladora:

```
Object resultado = Controlador.invoca(7, "processaNotasAte", hoje);
```

Apesar de funcionar, essa abordagem apresenta diversas desvantagens, além de claramente não ter um tratamento adequado aos recursos caros, nem um bom tratamento de suas exceções. Além disso, o desenvolvedor se torna responsável por código de infraestrutura que trabalha com `Sockets`, protocolos, exceptions e muito mais. Apesar de, nesse caso, não ser tão complicado manter um protocolo bem definido, tratar corretamente as exceptions no caso de o servidor cair, timeouts e outros detalhes são problemas de infraestrutura que levam bastante tempo para se resolver corretamente. O código ingênuo vai rapidamente se mostrar insuficiente.

Seria mais interessante se a aplicação obtivesse acesso a uma simples referência, que pudesse ser utilizada para invocar métodos em um objeto que está na aplicação servidora em outra JVM. Deseja-se invocar um método remotamente:

```
Double valor = empresa.processaNotasAte(new Date());
```

Para isto, é possível isolar uma nova camada que delega as invocações, extraindo uma interface e implementando a classe `Controlador`. Mesmo assim, todos os mé-

todos da implementação de `Empresa` passam a fazer um simples papel de *proxy*, delegando a requisição para o servidor remoto.

```
public interface Empresa {
    Double processaNotasAte(Date data);
}
public class EmpresaProxy implements Empresa {
    private final Long id;

    public EmpresaProxy(Long id) {
        this.id = id;
    }

    public Double processaNotasFiscaisAte(Date data) {
        try {
            Controlador ctrl = new Controlador();
            return (Double) ctrl.invoca(id, "processaNotasAte", data);
        } catch(IOException ex) {
            throw new RuntimeException("Erro na invocacao remota", ex);
        } catch(ClassNotFoundException ex) {
            throw new RuntimeException("Erro na invocacao remota", ex);
        }
    }
}
```

Um objeto que pode ser invocado de outra JVM é do tipo remoto. Neste caso, a API permite acessá-lo de maneira transparente, sem que o desenvolvedor que faz tais invocações tenha de codificar muito diferente de como faz com invocações locais. Às vezes, tal transparência pode trazer consequências negativas, como quando o desenvolvedor executa muitas chamadas remotas sem perceber Figura 4.2.

Remotabilidade é um dos *aspectos* que podem aparecer numa aplicação e deseja-se gastar o menor tempo possível com isso. Esses objetos que fazem o papel de delegadores de requisições para um objeto remoto são conhecidos, neste contexto, como *stubs* ou *proxies*, até aqui gerados programaticamente pelo desenvolvedor.

Para facilitar o trabalho e permitir que essas características sejam aplicadas a diversas classes distintas, é possível utilizar proxies dinâmicas do Java. Este é um recurso que permite criar, em tempo de execução, o *bytecode* de uma classe apenas em memória, que implementa determinadas interfaces, porém, toda invocação a esses objetos passará por um `InvocationHandler`.
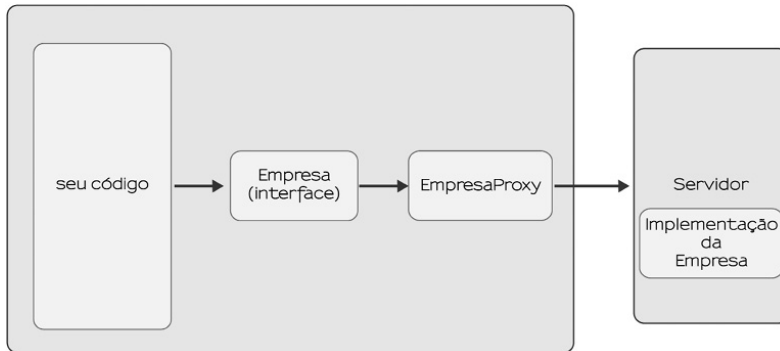
Figura 4.2: Aplicação usando um proxy para acessar o objeto remoto

```
public class RemoteProxy implements InvocationHandler {

    private final Long id;

    private RemoteProxy(Long id) {
        this.id = id;
    }

    public Object invoke(Object proxy, Method m, Object[] args) {
        try {
            return new Controlador().invoca(id, m.getName(), args);
        } catch(IOException ex) {
            throw new RuntimeException("Erro na invocação remota", ex);
        } catch(ClassNotFoundException ex) {
            throw new RuntimeException("Erro na invocação remota", ex);
        }
    }
}

public class Proxies {
    public static <T> T getRemoto(Long id, Class<T> type) {
        return (T) Proxy.newProxyInstance(
        this.getClass().getClassLoader(),
        new Class[] {type},
        new RemoteProxy(id));
    }
}
```

Fica simples acessar remotamente qualquer um dos nossos objetos:

```
Empresa remoto = Proxies.getRemoto(15L, Empresa.class);
Double valor = remoto.processaNotasAte(new Date());
```

Como o objeto referenciado pela variável `remoto` foi gerado dinamicamente, o nome da sua classe em tempo de execução será `Proxy$N`. Esta classe implementa as interfaces passadas como argumento, neste caso, `Empresa`. Os seus métodos, que estão num bytecode apenas em memória, delegam todas as invocações para o `InvocationHandler` dado. A Figura 4.3 ilustra o processo de invocação ao método remotamente através de uma proxy dinâmica.

A invocação `remoto.processaNotasAte(new Date())` só mostra que é remota pelo nome que foi dado para a referência à proxy, deixando a remotabilidade transparente. O código do `Controlador` é invocado internamente pela proxy. É comum encontrar diversos frameworks, como o RMI e o Hibernate, que vão tirar proveito dessas proxies dinâmicas, escondendo todo o código difícil e explícito que seria necessário para realizar tais tarefas. Um ponto negativo é exatamente essa transparência, que pode fazer com que o desenvolvedor não se atenha ao excesso de uso de recursos caros, como conexões ao banco e invocações remotas.

Proxies criadas através dessa API não podem estender classes, mas somente implementar interfaces. Existem alternativas para contornar esse limite, que vão gerar bytecode em tempo de execução, estendendo determinada classe e até mesmo alterando modificadores de acesso, como o *Javassist*, *cglib* ou *ASM* Figura 4.3.
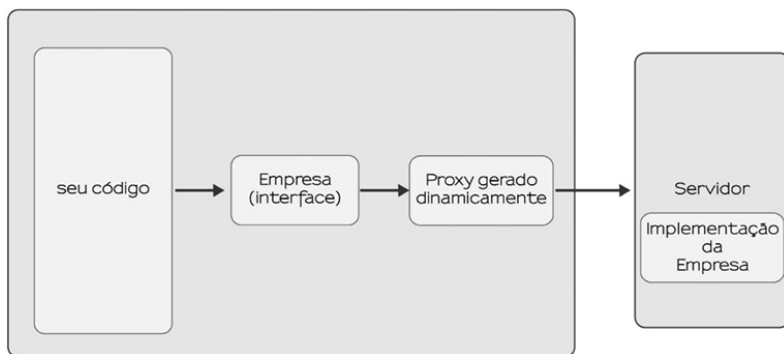
Figura 4.3: Aplicação utilizando o proxy dinâmico para acessar um objeto remoto

Apesar de ser um recurso poderoso, a manipulação de bytecode deve ser pon-

derada, assim como um design através de interfaces é preferível ao uso pesado de reflection. Ela vai ser frequentemente aplicada por diversos frameworks, que trabalham para ajudar sua infraestrutura, e conhecer essa capacidade ajuda a entender melhor a ferramenta, otimizá-la e enfrentar com mais facilidade as estranhas stack traces que podem surgir. O Hibernate, por exemplo, vai utilizar as proxies dinâmicas e manipulação de bytecode para fazer seu mecanismo de lazyness.

A programação orientada a aspectos (AOP), que ganhou evidência no começo dos anos 2000 através do AspectJ e do AspectWerkz, costuma ser implementada através da manipulação de bytecode. Hoje, apesar de toda expectativa, encontra-se mais num nicho bem específico, utilizado internamente aos containers. O JBoss, por exemplo, tem seu microkernel baseado em seu próprio projeto JBossAOP. O Spring também oferece uma biblioteca, a SpringAOP, para poder definir seus aspectos.[108]

Capítulo 5

Testes e automação

Equipes de desenvolvimento de software comumente evitam a atividade de testes, seja pela justificativa de um atraso na entrega ou pelo alto custo de realizá-los de forma manual. A não execução desta atividade diminui a qualidade final do software, que, na maior parte das vezes, apresentará problemas durante sua operação, que serão de difícil detecção e correção.

É impossível garantir que um software não possua falhas. O desenvolvedor deve assegurar a qualidade do software e, para que isso aconteça de maneira sustentável, é necessário criar e automatizar testes.

Existem muitos níveis de testes, e cada um deles entrega um tipo de feedback diferente para o desenvolvedor. O teste de sistema, por exemplo, garante a qualidade externa, verificando se a funcionalidade está de acordo com o requisitado pelo cliente. Já o teste de unidade ajuda o desenvolvedor a garantir a qualidade interna do código, dando feedback sobre o design dos módulos e permitindo uma manutenção com menor custo.

Este capítulo discute técnicas, vantagens e dificuldades de testes em diferentes

níveis, bem como alternativas para automatizá-los.

5.1 Testes de sistema e aceitação

Um software que funcione apenas na máquina de desenvolvimento não agrega valor para o cliente. É necessário algum recurso que garanta boa parte do correto funcionamento do código e que dê ao desenvolvedor o feedback sobre as versões de seu sistema em uma janela de tempo aceitável.

No mercado tradicional de desenvolvimento, são descritas, em longos documentos, chamados manuais de teste, baterias a serem executadas manualmente. Os testes que indicam se o comportamento do sistema não mudou e continua funcionando conforme o esperado são os testes que adicionam valor para o cliente.

O primeiro passo na quebra dessa abordagem tradicional é a automação deste procedimento, para remover erros humanos, diminuindo o tempo de feedback de cada mudança efetuada pelos desenvolvedores.

Testes que atravessam o sistema inteiro, desde a interface com o usuário até o backend de sua aplicação, são chamados *end-to-end*. Em aplicações Web, são aqueles que utilizam ferramentas como o *Selenium* para acessar diretamente, através de um navegador, o conteúdo das páginas do sistema. Com este resultado, são conferidas as expectativas através das asserções (através de `assert` e `verify`). O exemplo a seguir mostra o uso direto da API do *Selenium 2*:

```
@Test
public void oProdutoFoiAdicionadoAoCarrinho() throws Exception {
    browser.get("http://localhost/produtos/livro_sobre_arquitetura");

    // quando o usuário preenche a quantidade
    WebElement form = browser.findElement(By.id("produto"));
    form.findElement(By.name("produto.quantidade")).sendKeys("2");

    // e clica em adicionar();
    form.submit();

    // então a listagem mostra que o produto foi adicionado
    List<WebElement> linhas = browser.findElements(By.tagName(‘‘tr’’));
    WebElement ultima = linhas.get(linhas.size() - 1);

    WebElement produto = ultima.findElements(By.tagName("td")).get(2);
    assertThat(produto.getText(), is("Livro sobre arquitetura"));
```

```
    WebElement qtd = ultima.findElements(By.tagName("td")).get(3);
    assertThat(qtd.getText(), is("2"));
}
```

A vantagem deste tipo de teste é garantir que não haverá uma regressão nas funcionalidades já existentes do sistema, conferindo a aparição de bugs no código, que já funcionava conforme o esperado. Basta uma nova execução dessa bateria de testes para saber se alguma funcionalidade previamente testada deixou de funcionar.

Apesar dessa vantagem, esses testes são lentos e passam pelo ponto mais frágil de uma aplicação Web, a interface com o usuário. Qualquer mudança nela costuma acarretar a quebra de diversos testes *end-to-end* de uma só vez, mesmo quando as regras de negócio permanecem corretas. A atualização constante desse código de testes é custosa e pode ocorrer a cada mudança na interface. Por este motivo, é importante encontrar o número ideal de testes *end-to-end* que garanta o comportamento, e não atrapalhe a manutenção e evolução do sistema.

Teste de serviços Web

Ao testar serviços Web, cuja interface é uma representação em, por exemplo, *XML* ou *JSON*, em vez de utilizar o *Selenium* para carregar um navegador, utiliza-se uma *API*, como o *Restfulie* ou o *Http Apache Commons*:

```
public void deveRetornarAsUltimas3Cotacoes() {
    adiciona(new Cotacao("PETR4", 30.3));
    adiciona(new Cotacao("PETR4", 32.1));
    adiciona(new Cotacao("PETR4", 31.3));
    String xmlEsperado =
        "<cotacoes><cotacao>31.3</cotacao><cotacao>32.1</cotacao>" +
        "<cotacao>30.3</cotacao></cotacoes>";
    assertThat(
            Restfulie.at("http://localhost/cotacoes/PETR4").get().getBody(),
            is(equalTo(xmlEsperado)));
}
```

Apesar de não existir um navegador ou interface gráfica envolvido, esses testes também são *end-to-end*, pois atravessam o sistema inteiro. Eles garantem a qualidade externa ao código, trabalhando com serviços Web, seja através de SOAP, POX ou REST.

Testes de aceitação

Quando os testes são escritos de maneira a descrever o comportamento do sistema, também são utilizados para garantir que este último esteja de acordo com os requisitos levantados junto ao cliente.[143] Esse tipo de teste, principalmente em XP, é chamado de testes de aceitação (*acceptance tests*, ou testes de aceite).

Portanto, nem sempre esses testes de aceitação precisam passar por todo o sistema, e nem todos precisam ser *end-to-end*. Como no caso de adotar o *Selenium* somente para garantir a integração entre a interface do usuário e o *front controller*, sem validar a lógica de negócios por trás. Por exemplo, em um sistema de compra on-line, um teste valida que a interface responde como o esperado, sem garantir que a compra seja realmente efetuada. Isto é, as asserções não verificam o resultado dessa operação como lógica de negócio (suas consequências dentro do sistema). Esse resultado de negócios já está coberto por outra bateria de testes, de aceitação.

Ferramentas como Cucumber e o JBehave são ótimas para criar testes de aceitação, mas, segundo alguns autores, não deveriam ser usadas para testar lógica de negócios através da UI.[128] Um teste de aceitação que utiliza tais ferramentas, mas sem acessar a interface, consiste em garantir todos os fluxos de compra, diminuindo seu custo de manutenção.

De acordo com a legibilidade do código escrito, o teste de aceitação pode até ser encarado como uma forma de documentação. O exemplo a seguir utiliza métodos auxiliares, aumentando a expressividade e facilitando sua leitura:

```
@Test
public void mostraQueOProdutoFoiAdicionadoAUmCarrinhoDeCompras() {
    Carrinho carrinho = aoAcessarOProduto("livro_sobre_java")
        .escolher(2).eAdicionarNoCarrinho();
    Item adicionado = carrinho.getUltimaLinha();
    assertThat(adicionado.getProduto(), is(equalTo("Livro de Java"));
    assertThat(adicionado.getQuantidade(), is(equalTo(2));
}
```

Os testes de aceitação garantem que o comportamento será o mesmo que o esperado pelo cliente, dando segurança para uma possível entrega do software.

Testando os limites do sistema

Requisitos não funcionais também podem ser garantidos através de testes automatizados, como quando em um sistema é necessário que o tempo de resposta para

determinadas requisições leve no máximo 4 segundos, com uma média de 2 segundos. Testes de performance podem ser escritos através de ferramentas simples, como clientes HTTP, *ab* ou *JMeter*. Ao testar os servidores de produção ou similares com uma carga equivalente a cenários considerados extremos (*teste de carga*), é possível validar outro requisito não funcional: a escalabilidade. Levando em consideração as duas variáveis, temos dados que permitem analisar a performance em função do número de acessos simultâneos. A automatização desses testes garantirá o feedback rápido sobre a qualidade nesses requisitos não funcionais.

Testes executados pelo ser humano, que adicionam surpresas no roteiro, explorando casos particulares (*corner cases*) que não foram vislumbrados pelos desenvolvedores são chamados de testes de exploração e eles continuam importantes. Uma falha percebida por um teste de exploração pode ser rapidamente automatizada através da criação de um novo teste de aceitação.

5.2 Teste de unidade, TDD e design de código

Todos os testes servem como garantia de regressão, uma vez que, qualquer mudança que quebre os comportamentos esperados, será descoberta ao executá-los. Mas nem todos os testes são feitos com este fim. Testes menores podem verificar o comportamento de pequenas partes do código, tipicamente uma classe ou um método, chamados, neste contexto, de unidade.

Um *teste de unidade* deve garantir que a execução daquele trecho mínimo de código esteja correta. Para isso, verifica-se o retorno de uma invocação, a interação do trecho com outras partes do sistema ou o estado do objeto. No exemplo a seguir, a classe Venda é utilizada para calcular o imposto a ser pago em uma venda, enquanto seu teste verifica de maneira isolada se o método `getImposto` funciona conforme o esperado.

```
public class Venda {

    private final Produto[] produtos;
    public Venda(Produto... produtos) {
        this.produtos = produtos;
    }

    public double getImposto() {
        double imposto = 0;
        for (Produto produto : produtos) {
```

```
            imposto += produto.getPreco() * 0.1;
        }
        return imposto;
    }
}

public class VendaTest {
    @Test
    public void deveCalcularDezPorcentoSobreOPrecoDosProdutos() {
        Produto livro = new Produto("Arquitetura e Design", 100);
        Produto computador = new Produto("Notebook", 2000);
        double imposto = new Venda(livro, computador).getImposto();
        assertEquals(210.0, imposto, 0.001);
    }
}
```

Verificar o retorno de uma invocação pode ser feito com uma simples asserção, mas conferir a interação entre componentes pode ser difícil, dependendo do design das suas classes.

O exemplo a seguir mostra um controlador Web que acessa a base de dados para agendar uma reunião, utilizando a abordagem tradicional de criar primeiro o código e depois o teste:

```
public class ClienteController {
    public String agenda(long id, Calendar horario) {
        Cliente cliente = ClienteDao.getInstance().busca(id);
        Reuniao reuniao = new Reuniao(cliente, horario);
        ReuniaoDao.getInstance().cria(reuniao);
        return "sucesso";
    }
}
```

Apesar de funcional e correto, o código anterior apresenta um alto acoplamento com as classes `ReuniaoDao` e `ClienteDao`. Ele também possui baixa coesão ao se preocupar tanto com a regra de negócios quanto com detalhes da conversão de `long` para `Cliente`. Ao criar um teste isolado para este *controller*, encontra-se algumas barreiras:

```
public class ClienteControllerTest {
    @Test
    public void deveSalvarReuniaoComOsDadosDoCliente() {
```

```
        Calendar horario = Calendar.getInstance();
        ClienteController cliente = new ClienteController();
        assertEquals("sucesso", cliente.agenda(15, horario));
    }
}
```

O teste anterior pode funcionar, mas não de maneira isolada. A invocação ao método agenda implica a chamada a busca e cria, métodos pertencentes a outras classes. Além disso, falta a verificação de como tais métodos foram executados corretamente, uma vez que o teste só garantiu o retorno.

Para testar esse código por completo, é necessário fazer asserções em outras partes do código. Isso indica que há um possível excesso de acoplamento com suas dependências. Não é possível testá-lo como uma unidade. Para sistemas legados e sem testes, o código a seguir é um primeiro passo de um processo aceitável, mas o próximo seria o da refatoração, visando a diminuição do acoplamento.

```
public class ClienteControllerTest {
    @Test
    public void deveSalvarReuniaoComOsDadosDoCliente() {
        Calendar horario = Calendar.getInstance();
        ClienteController cliente = new ClienteController();
        assertEquals("sucesso", cliente.agenda(15, horario));
        Cliente cliente = ClienteDao.getInstance().busca(15);
        Reuniao reuniao = ReuniaoDao.getInstance().buscaUltima();
        assertEquals(cliente, reuniao.getCliente());
    }
}
```

Para possibilitar a escrita do teste de maneira isolada, o primeiro passo é tornar explícitas as dependências de uma classe, recebendo-as de alguma maneira, conforme os princípios de injeção de dependências e do *Good Citizen*[146] Para isso, é necessário remover a invocação direta ao método `getInstance`.

```
public class ClienteController {

    private final ClienteDao clientes;
    private final ReuniaoDao reunioes;

    public ClienteController(ClienteDao clientes, ReuniaoDao reunioes){
        this.clientes = clientes;
```

```
        this.reunioes = reunioes;
    }

    public String agenda(long id, Calendar horario) {
        Reuniao reuniao = new Reuniao(clientes.busca(id), horario);
        reunioes.cria(reuniao);
        return "sucesso";
    }

}
```

Recebendo tais dependências, elas podem ser facilmente trocadas durante a execução do teste:

```
public class ClienteControllerTest {

    @Test
    public void deveSalvarReuniaoComOsDadosDoCliente() {
        ClienteDao clientes = mock(ClienteDao.class);
        ReuniaoDao reunioes = mock(ReuniaoDao.class);

        Cliente cliente = new Cliente();
        when(clientes.busca(15)).thenReturn(cliente);

        Calendar horario = Calendar.getInstance();
        ClienteController controller =
            new ClienteController(clientes, reunioes);
        assertEquals("sucesso", controller.agenda(15, horario));
        verify(reunioes.cria(new Reuniao(cliente, horario)));
    }
}
```

O teste anterior cria um `ClienteController` e passa instâncias de `ClienteDao` e de `ReuniaoDao`. Mas, desta vez, os objetos são outros; não são implementações concretas daquelas classes, e sim objetos que simulam o comportamento das mesmas. Estes objetos são chamados *mocks*. Assim, é finalmente possível testar o controlador sem a interferência do resto do sistema.

Essa refatoração diminuiu o acoplamento da classe `ClienteController` com os DAOs, tirando a responsabilidade de gerenciar o ciclo de vida das dependências. O controlador agora não depende de uma implementação concreta dos DAOs, mas

apenas da interface pública dos mesmos. O acoplamento é bem menor e mais explícito.

Tanto `ClienteDao` quanto `ReuniaoDao` poderiam ser transformados em interfaces (*extract interface*), mais estáveis que classes concretas, conforme visto no capítulo sobre orientação a objetos; assim, todo código e testes que dependem delas sofreriam menos mudanças, tornando-os mais gerenciáveis. Com as interfaces, o controlador permitiria que qualquer implementação fosse utilizada, removendo o acoplamento direto às implementações dos DAOs.

Se o **design possui alto acoplamento, será praticamente impossível testar seu código através de testes de unidade**[78]. Por este mesmo motivo, métodos estáticos e estado global dificultam os testes de unidade, assim como todo tipo de busca explícita por uma dependência aumenta o acoplamento (singletons ou factories através de métodos estáticos e new indiscriminados).

Uma vantagem dos testes de unidade sobre outros tipos é que, além de garantir que antigos bugs não se repitam no futuro, eles dão feedback sobre o acoplamento e a coesão do código. Mas receber tal feedback apenas após escrever muitas linhas de código do sistema pode ser tarde, pois, nesse momento, a refatoração e a melhoria do design podem ser difíceis e apresentar um custo mais caro.

O risco em testar depois, isto é, criar primeiro a funcionalidade e só então o teste, é ser influenciado pelo vício da mente do desenvolvedor após a escrita do código. Nesta situação, o teste é escrito para comprovar que aquilo que foi desenvolvido está de acordo com o que o programador entendeu. Se a funcionalidade foi implementada de maneira errada, é comum que o teste também esteja. Ao invés de garantir que o código funciona adequadamente, ele garante que um bug continua sendo um bug, enquanto dá a sensação de que está funcionando. É fácil também deixar sem testes alguns dos possíveis caminhos que o código pode seguir, como não testar uma condição `else` ou uma `exception`.

Até este instante, a criação dos testes foi feita após a implementação de uma funcionalidade. Invertendo a ordem, ou seja, escrevendo primeiro o código de teste e depois a implementação (*test-first*), recebe-se feedback constante sobre o design do sistema, além de tornar obrigatória a existência de pelo menos um teste para cada funcionalidade. Desta forma, um desenvolvedor atento se policia para implementar somente aquilo que seu teste precisa para passar.

Em uma lógica de negócios simples, que envolve dois possíveis caminhos (`if` e `else`), essa abordagem obriga o desenvolvedor a escrever um teste para cada uma das duas situações.

Uma consequência da existência de testes para todas as funcionalidades é a alta cobertura do código (*code coverage*). Ela serve como métrica para indicar que as partes com baixa cobertura precisam ser analisadas com cautela quanto ao seu funcionamento. Mas a existência de um teste não garante que ele funcione corretamente, como visto no caso dos testes escritos após a implementação, ou, ainda, quando o desenvolvedor testa erroneamente uma funcionalidade. A Figura 5.1 mostra parte do relatório de cobertura de um projeto.

Um teste descreve como o código será utilizado. Quanto mais complicado o código de um teste, mais problemático está o design adotado. Por exemplo, um teste que requer diversas inicializações de dependências indica um alto acoplamento, enquanto se ele envolve muitas verificações distintas, há possíveis problemas de coesão.

br.com.caelum.vraptor.converter	100%	23/23	95%	95/100
br.com.caelum.vraptor.util	100%	6/6	92%	33/36
br.com.caelum.vraptor.ioc	100%	6/6	89%	34/38
br.com.caelum.vraptor.interceptor	93%	14/15	89%	132/148
br.com.caelum.vraptor.eval	100%	1/1	88%	23/26
br.com.caelum.vraptor.ioc.guice	92%	24/26	88%	67/76
br.com.caelum.vraptor.http.iogi	100%	6/6	88%	14/16

Figura 5.1: Exemplo de relatório de cobertura de linhas e branches.

É fundamental que o desenvolvedor domine boas práticas de design para que seja capaz de interpretar o feedback de um teste. Antes de escrever a implementação de uma funcionalidade, a criação de um teste dá liberdade para que o desenvolvedor escreva o que acredita ser um bom design, sem se preocupar com os detalhes de implementação.

Suponha que uma aplicação Web tenha a necessidade de finalizar uma venda a partir de um pedido já feito pelo usuário, através da gravação no banco de um pedido, formado por produto e preço. Além disso, é necessário também enviar uma mensagem para outro sistema que será responsável pelo envio do produto para o cliente e diminuir a quantidade do produto no estoque.

Criando o teste primeiro é obtido um código desejado de execução:

```
public class VendasTest {
```

```
    @Test
    public void deveFinalizarVendaSalvandoEnviandoEDiminuindoEstoque(){
        VendaDao dao = mock(VendaDao.class);
        Estoque estoque = mock(Estoque.class);
        ClienteHttp httpClient = mock(ClienteHttp.class);
        Endereco destino = new Endereco("Rua Vergueiro 3185, SP");
        Produto livro = new Produto("Arquitetura e Design", 100);
        Produto computador = new Produto("Notebook", 2000);
        Pedido pedido = new Pedido(destino, livro, computador);
        Vendas vendas = new Vendas(dao, estoque, httpClient);
        Venda venda = vendas.vende(pedido);
        // salvar a venda
        verify(dao.salva(venda));
        // diminuir do estoque os produtos da venda
        verify(estoque.diminui(venda.getProdutos()));

        // enviar a venda para outro sistema
        String xml = new XStream().toXml(venda);
        verify(httpClient.post(xml));
    }
}
```

Não há problemas na existência de mais de um assert por método de teste, desde que todos estejam garantindo um único comportamento. Entretanto, o teste acima indica que a abordagem levaria a uma classe Vendas que possui diversas responsabilidades; além de criar e salvar uma Venda, ela também é responsável por diminuir o estoque, serializá-la e enviá-la para o outro sistema.

O custo de refatorar a classe `Vendas` ainda é baixo, já que a classe não existe. Um desenvolvedor atento perceberia a falha de design ao escrever esse nome do método de teste. Tantas palavras no nome de um método, ainda mais concatenadas com "*es*", indicam que há muitas responsabilidades. Refatorá-lo até mesmo antes da implementação seria um passo fundamental para aumentar a coesão dessa classe.

O código de teste começa a indicar que há muitos passos envolvidos em um processo de venda, e que deve ser interessante quebrar as responsabilidades dessa classe `Vendas` em pequenas classes e, depois, juntá-las novamente. A principal responsabilidade desta classe é gerar uma Venda propriamente dita. O resto pode ser composto nela, através de *observers*[80], analogamente à discussão sobre composição no capítulo sobre OO.

O método vende passará a ter uma única responsabilidade, a de gerar a venda baseada no pedido. Após essa refatoração, o seguinte conjunto de testes é obtido:

```
public class VendasTest {

    // método auxiliar para gerar um pedido
    private Pedido umPedido() {
        Endereco destino = new Endereco("Rua Vergueiro 3185, SP");
        Produto livro = new Produto("Arquitetura e Design" , 100);
        Produto computador = new Produto("Notebook" , 2000);

        return new Pedido(destino, livro, computador);
    }

    @Test
    public void deveGerarUmaVenda() {
        Vendas vendas = new Vendas();
        Venda venda = vendas.vende(umPedido());

        assertEquals(2100.0, venda.getTotal(), 0.00001);
    }

}
```

A implementação do método vende para o teste anterior é bem simples:

```
public class Vendas {

    public Venda vende(Pedido pedido) {
        return geraVendaAPartirDe(pedido);
    }

    private Venda geraVendaAPartirDe(Pedido pedido) {
        // gera uma venda baseada nas regras de negócio
    }
}
```

Agora que a venda já foi gerada, é necessário adicionar alguns passos de um processo. Caso tais comportamentos fossem implementados dentro desta classe, o resultado seria a baixa coesão. Portanto, quando aplicável, quebra-se o comportamento em diversos passos. A classe Processo funciona como uma lista de Passos:

```
public interface Passo {
    void processa(Venda venda);
}

public class Processo implements Iterable<Passo> {

    private final List<Passo> passos;

    public Processo(Passo... passos) {
        this.passos = Arrays.asList(passos);
    }

    public Iterator<Passo> iterator() {
        return passos.iterator();
    }
}
```

A classe `Vendas` notificará a lista dos próximos passos a serem executados com a `Venda` recém-criada. É necessário adicionar um teste que garanta tal comportamento:

```
@Test
public void devePassarAVendaGeradaPorTodoOProcessoDeVenda() {
    Passo passo1 = mock(Passo.class);
    Passo passo2 = mock(Passo.class);

    Processo processo = new Processo(passo1, passo2);
    Vendas vendas = new Vendas(processo);
    Venda venda = vendas.vende(umPedido());

    verify(passo1.processa(venda));
    verify(passo2.processa(venda));
}
```

Repare novamente no baixo acoplamento: a implementação da classe `Vendas` não sabe quais serão os passos executados.

```
public class Vendas {

    private final Processo processo;

    public Vendas(Processo processo) {
```

```
        this.processo = processo;
    }

    public Venda vende(Pedido pedido) {
        Venda venda = geraVendaAPartirDe(pedido);

        for (Passo passo : this.processo) {
            passo.processa(venda);
        }

        return venda;
    }

    private Venda geraVendaAPartirDe(Pedido pedido) {
        // gera uma venda baseada nas regras de negócio
    }
}
```

A criação dos passos segue os detalhes necessários para cada trecho de lógica, sendo que cada responsabilidade que o desenvolvedor julgar suficientemente desconexa das outras terá sua própria classe:

```
public class SalvaVenda implements Passo {

    private final VendaDao dao;

    public SalvaVenda(VendaDao dao) {
        this.dao = dao;
    }

    public void processa(Venda venda) {
        this.dao.salva(venda);
    }
}

public class ReduzEstoque implements Passo { ... }

public class EnviaVendaParaOutroSistema implements Passo { ... }
```

O design final oferece maior coesão e menor acoplamento, através da adoção de conceitos de injeção de dependências e composição de comportamentos através de

uma cadeia de passos a serem seguidos. A solução apresentada acima é uma das possíveis maneiras de resolver o problema de baixa coesão, e cabe ao desenvolvedor encontrar a mais adequada ao seu problema.

Abordagens como esta utilizam também a prática de passos tão pequenos quanto façam sentido, mas não necessariamente o menor de todos (o menor de todos é chamado *baby step*), para ajudar o desenvolvedor a não pular passos.

Perceba que os testes de unidade e o feedback constante que eles dão sobre o design guiaram o programador no desenho das classes acima, diferente do feedback fornecido pelos testes *end-to-end*. O processo de escrever testes antes da implementação e usar o feedback deles para guiar o processo é conhecido por *Test-Driven Development* (*TDD*)[15]. Esse impacto no design é tão grande que muitos utilizam a sigla TDD para *Test-Driven Design* [7] (Figura 5.2).

O design da classe emerge de como o desenvolvedor deseja utilizá-la. Dependerá do conhecimento do desenvolvedor a escolha dos padrões e boas práticas a serem aplicados. No exemplo anterior, o design fugiu de modelos anêmico[61], adotou o uso de um repositório[30] e injeção de dependências através do construtor.

TDD é uma ferramenta, e sozinha não garante resultado, mas permite alcançá-lo quando em conjunto com outras práticas. Testes ajudam no design mostrando basicamente os problemas ao programador; um teste complicado demais é um indício de possíveis falhas desse design[78] [127]. Um código fácil de testar tende a apresentar um bom design. Michael Feathers possui uma excelente frase que resume a ideia: "*Existe uma grande sinergia entre testabilidade e um bom design*"[58].
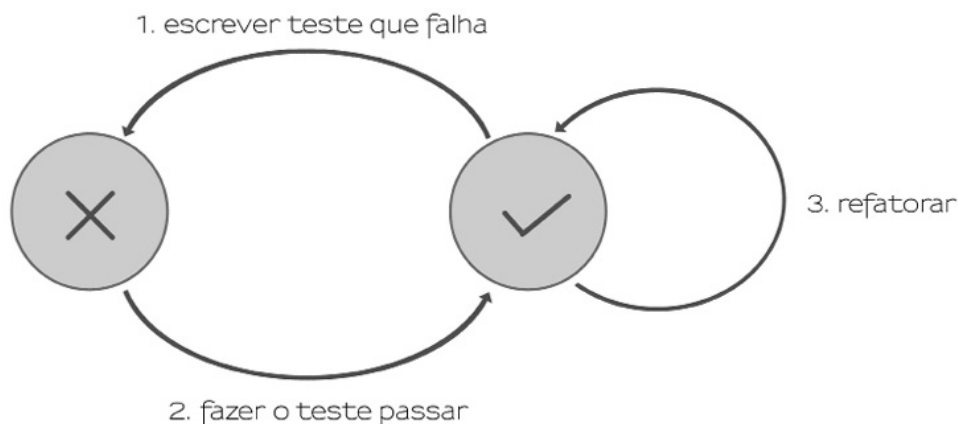
Figura 5.2: O processo de test-first, implementação e refatoração

Testar o comportamento é a chave dos testes, que devem somente validar o esperado, e não como o resultado esperado é atingido. Ao criar os testes, executa-se uma ação e garante-se que o comportamento daquela unidade ou funcionalidade em questão é o esperado.

A ideia de escrever os testes de unidade antes da implementação é tão interessante que pode ser estendida para os testes de sistema. A técnica, conhecida como *Acceptance Test-Driven Development* (ou *ATDD*), sugere que o programador crie primeiro um teste de sistema antes de começar a implementação. Ela garante que toda equipe tenha o mesmo entendimento sobre o que deve ser feito, já que os testes de aceitação são definidos antes mesmo de iniciar a implementação.

5.3 Testando a integração entre sistemas

É necessário garantir também que a comunicação entre componentes internos da aplicação e outros sistemas funcione, como, por exemplo, uma camada de DAO que acessa um banco de dados. Enquanto os testes *end-to-end* testam uma aplicação inteira e testes de unidade, o comportamento isolado, os de integração (*integration tests*) verificam uma parte da pilha de execução.

DAOs comumente recebem um `EntityManager` ou similar em seu construtor, e o utilizam para enviar statements para um banco. Ao começar pelo teste, é possível utilizar *mocks* para verificar se a query esperada é passada para o manager, como no

exemplo a seguir; um DAO chamado `ClienteDao`, que possuirá um método responsável por retornar os cinco clientes incluídos mais recentemente e que estão marcados como importantes:

```
@Test
public void deveBuscarOsCincoMaisRecentesEImportantes() {
    EntityManager manager = mock(EntityManager.class);
    Query query = mock(Query.class);
    List<Cliente> resultados = new ArrayList<Cliente>();
    String jpql = "select from Cliente where importante=true " +
        "order by dataDeCriacao";
    when(manager.createQuery(jpql)).thenReturn(query);
    when(query.setMaxResults(5)).thenReturn(query);
    when(query.list()).thenReturn(resultados);

    ClienteDao dao = new ClienteDao(manager);
    List<Cliente> clientes = dao.cincoMaisRecentesEImportantes();
    assertThat(clientes, is(equalTo(resultados)));
}
```

Após criado o teste, adiciona-se o método no DAO, mas o teste verifica se os métodos de `EntityManager` e `Query` são invocados; portanto, a implementação será baseada nesta informação, com os exatos parâmetros esperados.

```
public class ClienteDao {
    private final EntityManager manager;

    public ClienteDao(EntityManager manager) {
        this.manager = manager;
    }

    public void adiciona(Cliente cliente) {
        manager.persist(cliente);
    }

    public List<Cliente> cincoMaisRecentesEImportantes() {
        String jpql = "select from Cliente where importante=true "
            + "order by dataDeCriacao asc";
        return manager
            .createQuery(jpql)
            .setMaxResults(5).list();
```

```
    }

    public List<Cliente> todos() {
        return manager.createQuery("from Cliente").list();
    }
}
```

Ao dizer que o método deve buscar cinco clientes e testar somente se uma `String` é passada para um outro método, o próprio nome do teste indica ao desenvolvedor que existe algo de errado. Não está sendo testado se o código busca os clientes, mas sim se ele passa um parâmetro.

Quanto mais o teste se assemelha ao código criado, mais as asserções validam a implementação, em vez do resultado esperado do comportamento. Caso o desenvolvedor tenha escrito a `String` de uma maneira sintaxicamente válida, mas retornando resultados incorretos, a implementação costuma ser feita com a mesma `String`, e o teste passa a ser um mero validador de que o que o desenvolvedor escreveu na implementação é o mesmo que esperava digitar, não adicionando valor à bateria.

No exemplo anterior, o desenvolvedor pensou em como será implementado o comportamento enquanto escrevia o teste, antes da implementação, cometendo um erro ao utilizar a `String` asc, em vez de desc. Neste cenário, o desenvolvedor é induzido a usar na implementação a mesma `String` do teste. Como ele não se preocupou em verificar a ordem, mas somente a `String`, este erro foi propagado para a implementação. Ao testar primeiro, é importante pensar no comportamento esperado, e não na implementação, para evitar situações como esta, em que a `String` utilizada no teste não garante o comportamento correto da funcionalidade.

Para evitar testes inúteis que somente refletem aquilo que foi digitado em componentes que envolvem a interação com outros sistemas, é necessário verificar que o comportamento da unidade seja o esperado. O código a seguir demonstra como verificar essa integração:

```
public class ClienteDaoTest {
    private EntityManagerFactory factory;
    private EntityManager manager;

    private ClienteDao clienteDao;

    @Before
    public void setUp() {
        factory = Persistence.createEntityManagerFactory("teste");
```

```
        manager = factory.createEntityManager();
        clienteDao = new ClienteDao(manager);
        manager.getTransaction().begin();
    }

    @After
    public void tearDown() {
        manager.getTransaction().close();
        manager.close();
        factory.close();
    }

    private cliente clienteImportante() {
        Cliente cliente = new Cliente();
        cliente.setImportante(true);
        return manager.persist(cliente);
    }
    return cliente;

    @Test
    public void deveBuscarOsCincoMaisRecentesEImportantes() {
        List<Cliente> esperados = new ArrayList<Cliente>();
        for (int i = 0; i < 5; i++) {
            esperados.add(clienteImportante());
        }

        // esperamos o mais recente primeiro
        Collections.reverse(esperados);
        List<Cliente> importantes =
            clienteDao.cincoMaisRecentesEImportantes();
        assertThat(importantes, is(equalTo(esperados)));
    }
}
```

A implementação da `JPA` já foi testada pelo seu fornecedor, mas o teste feito aqui não é para garantir o funcionamento, por exemplo, da ::JPA *com o* MySQL*, mas sim que a query criada dentro da `String` foi feita de tal maneira que corresponde à lógica de negócios que desejamos alcançar. Um dos pontos fundamentais da importância de tal teste está no fato de que a `String` da query possui* branches::, fluxos distintos, de lógica. Ferramentas de cobertura de código não detectam esses detalhes, e entendem que a query é uma `String` como outra qualquer. Porém, ela possui diversas regras

embutidas que podem ter sido escritas de maneira errada; portanto, o teste para verificar que a query não devolve nada além do esperado é fundamental.

Um teste de integração bastante discutido diz respeito ao método adiciona descrito anteriormente. Ele delega uma requisição para o manager e, à primeira vista, parece ser desnecessário testá-la. Caso não haja um teste que passe pela integração com outro sistema e o desenvolvedor anotar a classe `Cliente` de maneira inválida, usando um tipo de campo que não é suportado no banco escolhido, os testes retornarão uma falsa sensação de que a configuração está válida e de que o sistema está consistente, quando na realidade não está.

Os testes de integração devem tocar a API que acessa os sistemas externos. Estes podem ser uma cópia do sistema de produção ou uma versão mais simples, como um grande *mock* ou ambiente de teste, mas é importante que ele se comporte exatamente como o sistema real ao qual será feita a integração quando em produção.

5.4 Feedback através de integração contínua

É arriscado recolher, somente no dia da entrega, informações sobre um novo bug, devido à mudança em parte do código. É necessário que esta informação apareça rapidamente para o desenvolvedor responsável. Entregar funcionalidades frequentemente permite receber um feedback rápido sobre o sucesso ou falha de cada mudança no projeto.

É também importante descobrir rapidamente que uma funcionalidade nova não atende às necessidades do cliente como ele imaginava no momento de sua concepção, se é necessário uma alteração ou até mesmo sua remoção.

A criação de um software é baseada na exploração das possibilidades de implementação que ele fornece para o cliente, e o único momento em que ele pode ter certeza sobre um caminho escolhido é durante seu uso. Tudo isso requer feedback rápido e contínuo de nossas mudanças na base de código.

Quanto mais tardio o feedback recebido pelos desenvolvedores a respeito de um bug ou funcionalidade que não atinge o objetivo, mais difícil lembrar o que, como e os motivos pelos quais foi tomada uma decisão no desenvolvimento que resultou no mesmo. Quanto mais tempo os desenvolvedores mantiverem suas alterações na base de código em sua própria máquina, sem enviá-las ao servidor de controle de versão (*version control system*) para integrá-las ao código de seus colegas, mais devagar será o feedback de suas decisões. Integrar frequentemente minimiza o tamanho dos *merges* que devem ser feitos, além de permitir receber feedback o mais rápido possível.

Para isto, devemos integrar as alterações, feitas sempre que julgar razoável – o que é um conceito subjetivo –, mas, quanto mais cedo, melhor.

O mercado tem adotado commits locais minimais, que envolvem passos pequenos de decisão, como uma refatoração simples e integração do código com o controle de versão assim que uma funcionalidade é completada, ou um bug corrigido. Pode-se levar mais ao extremo, efetuando a integração sempre que algo é alterado e não quebra nenhuma funcionalidade preexistente. Integrar nessa frequência, sempre e o mais rápido possível, tentando minimizar a influência de janelas longas sem integração, é chamado integração contínua (*continuous integration*).

É importante que a maior parte possível dos testes seja executada a cada commit, para que a informação do impacto das alterações na base de código venha rapidamente, permitindo ao desenvolvedor agir de maneira adequada para corrigir falhas e bugs que possam ter se introduzido. James Shore, em seu livro sobre diversas práticas de XP[173], cita 10 minutos como um tempo bom para tal retorno de informação.

Nossa experiência indica que esse tempo varia entre cada projeto, equipe e momento do seu desenvolvimento. Quanto mais baixo melhor, mas é necessário, ao mesmo tempo, escolher um conjunto de testes que garantam um feedback de qualidade e não demore mais do que o desejado.

O processo inteiro, desde a compilação até o deploy, pode ser quebrado em diversas fases, descrito em um *build pipeline* (Figura 5.3). Um exemplo de *pipeline* envolve a fase inicial de compilação e verificação de métricas de qualidade; depois, a execução de testes de unidade; terminando com uma terceira fase automática de testes *end-to-end*. A partir deste instante, esse pipeline poderia continuar sua execução através de um clique que indica o deploy para *homologação* ou *produção*.

Figura 5.3: Possível representação de um build pipeline com 5 etapas.

Integração contínua e branches

Uma alternativa à integração contínua é a criação de um branch para cada funcionalidade que está sendo desenvolvida (*feature branches*). Durante todo o tempo em que cada branch vive separado, a integração deixa de ser contínua, e o feedback que os desenvolvedores recebem sobre suas alterações na base de código se torna cada vez mais lento, além dos problemas de *merge* inerentes a longos branches.

Em outra situação, quando extremamente necessário executar o commit de uma funcionalidade parcial, as funcionalidades podem ser mantidas todas dentro de um único branch através de técnicas, como a de configurações que as ativam ou desativam (*feature toggle*), ou, ainda, opções em tempo de compilação que removem ou adicionam partes ao sistema[178][179]. Evan Bottcher cita que a criação de diversos *branches* paralelos vai contra a propriedade coletiva do código, uma vez que equipes, trabalhando em isolamento, não compartilham o conhecimento do trabalho executado. Uma vez que o processo de merge seria mais penoso, os desenvolvedores podem acabar por evitar realizar refatorações, resultando em um débito técnico maior em relação ao design da aplicação.[22]

A adoção de um sistema de controle de versão, a integração contínua, commits minimalistas, *merges* pequenos e IDEs com funcionalidades inteligentes de refatoração são ferramentas que permitem ao desenvolvedor efetuar mudanças na base de código sem medo de possíveis consequências negativas, pois, com o feedback rápido, evita-se enviar algum código que não funcione para um sistema em produção.

Build automatizado

Existem diversas ferramentas de *build automatizado*, que facilitam o processo de execução de seus testes, uma vez que sigamos o conceito de integração contínua.

oDentre elas, destacam-se o Cruise.rb (e suas variações, como o Cruise Control) e Jenkins, antigo Hudson, com código open source, enquanto o Cruise e o TeamCity são exemplos de produtos pagos. James Shore cita que, para configurar o projeto para rodar na máquina de build, devemos automatizar o processo de configuração através de scripts[173]. A própria máquina de build também pode ser configurada através de scripts, facilitando a criação de diversas máquinas de build e possibilitando distribuí-lo.

Quando o projeto cresce, gargalos no tempo de execução do build surgem, podendo passar de um limite aceitável. É possível paralelizar a execução do build, distribuindo o processo entre diversas máquinas. Poucos servidores de build suportam

nativamente distribuir e gerenciar as tarefas. O objetivo de paralelizar partes do processo entre diversas máquinas é fazer com que o feedback para os desenvolvedores seja recebido mais rapidamente, mesmo em projetos grandes.

O desafio ao adotar as práticas de testes automatizados e o processo de TDD é encontrar uma escolha de um conjunto de tipos de testes distintos que ajude no processo de design do seu código, maximize as garantias de regressão e o correto funcionamento, além de minimizar o tempo de feedback.

Entrega contínua

Para entregar funcionalidades ao cliente é necessário colocar o produto em um sistema de produção, um processo chamado ::deploy:;. É possível encontrar aplicações onde ele é totalmente manual, suscetível a diversos tipos de falha humana. As abordagens frequentemente encontradas envolvem a existência de um único responsável pelo deploy, que utiliza um arquivo de texto onde estão descritos os passos a serem executados.

Em casos mais graves, o roteiro é seguido por alguém que não tem vínculo algum com o projeto, somente a tarefa de segui-lo, sem entender o funcionamento da aplicação. Isso pode acarretar um grave aumento de custo, já que possíveis falhas parciais no processo de deploy podem deixar os dados ou a aplicação em um estado inconsistente. Nesse caso, somente os desenvolvedores seriam capazes de entender o motivo e as consequências de se prolongar essa situação por muito tempo, implicando maior esforço humano para corrigi-los. Esse tipo de deploy costuma ser feito às sextas-feiras à noite, com uma frequência baixíssima, tentando evitar os picos de acesso aos sistemas e minimizar possíveis problemas, mas, mesmo assim, por diversas vezes os desenvolvedores são acionados para passar o fim de semana corrigindo um deploy falho.

Diminuir o intervalo entre entregas, criando um processo de entrega contínua (*continuous delivery*, ou, ainda, entrega frequente)[101] [177] é fundamental para melhorar a qualidade do produto desenvolvido. Para entregar frequentemente, é necessário que o processo de deploy não implique custos altos, como, por exemplo, o de seres humanos interpretarem arquivos que descrevem o processo de deploy. Por serem executados com maior frequência, é necessário que tomem pouco tempo de execução e exijam pouca ou nenhuma interação humana, permitindo que qualquer um os execute e efetue correções mais cedo no processo de desenvolvimento do software.

Para minimizar os erros do processo de deploy, as tarefas, cuja responsabilidade

era de um ser humano, devem ser agora automatizadas. Toda execução de linha de código, cópia de arquivo, back-up dos dados e mudanças de schema do banco de dados, entre outros, devem ser executados através de scripts.

São inúmeros os projetos cujo processo envolve somente a cópia de artefatos e arquivos de configuração distintos para o ambiente de produção. E, em tais casos, automatiza-se com a criação de scripts, invocando comandos de transferência de arquivos como scp ou sftp, que serão recebidos pelo servidor e colocados "no ar". Opções em outras linguages, como Ruby, são a utilização de ferramentas, como o capistrano, que possuem foco em automatização de tarefas em si. O exemplo a seguir mostra um script que, antes de executar o deploy de um arquivo *war*, extrai um back-up do banco de dados e da versão atual em produção da aplicação.

```
#!/bin/sh
if [ -f version ]; then
DUMPVERSION=`cat version`
else
DUMPVERSION=0
fi

DUMPVERSION=$((DUMPVERSION + 1))
echo ‘‘$DUMPVERSION’’ > version

mysqldump -u usuario -p senha sistema > snapshots/dump_$DUMPVERSION
cp webapps/application.war snapshots/application_$DUMPVERSION
cp latest_application.war webapps/application.war
bin/restart.sh
```

Processos de deploy mais complexos envolvem, por exemplo, a migração de estruturas de um banco de dados relacional, a criação de novos índices de busca, a atualização de dados agrupadores parciais, utilizados para relatórios de *business inteligence::, e podem ser automatizados com ferramentas específicas que permitem a configuração de um processo de deploy (*deploy pipeline::), como com scripts ant junto com suas tasks para deploy em servidores de aplicação[214], ou builds completamente controlados pelo maven.

Ferramentas especializadas permitem que equipes de desenvolvimento e operações trabalhem mais próximas, gerenciando a configuração de sistemas operacionais e softwares (*configuration management*)[101]. Esse gerenciamento pode ser alcançado com ferramentas como Puppet, Chef e CFEngine, criando scripts que declaram como deve ser feita a configuração de um conjunto de máquinas, desde seu sistema

operacional, passando pelos servidores de aplicação, load balancers, etc., tentando aproximar o processo de desenvolvimento, a entrega e a operação de software.[152]

Uma aplicação Web típica pode declarar em sua configuração algum software para *load balancing*. Este, por sua vez, é configurado para apontar para as máquinas com servidores Web, como Apache https, Jetty ou Glassfish. E o servidor Web, por sua vez, aponta para máquinas com bancos de dados que rodam MySQL ou outros. Toda essa configuração foi feita pelo mesmo script.

Todas essas receitas devem ser guardadas com controle de versão. Dessa forma, a qualquer instante, um desenvolvedor pode replicar qualquer versão histórica do sistema em um conjunto separado de máquinas. E ainda, restaurando um backup do banco de dados, pode simular bugs que aconteceram em versões específicas do sistema.

Adotando tal prática, é possível replicar o ambiente de produção com um custo muito baixo; basta executar o script para que seja criado um novo ambiente de homologação mais próximo ao de produção, ajudando a encontrar bugs mais cedo. Ambientes de desenvolvimento também podem ser automatizados da mesma maneira, um processo que ajuda quando as equipes possuem alta rotatividade de desenvolvedores.

Para permitir a execucão de um deploy em poucos minutos através de um único clique, automatiza-se tudo o que for possível, facilitando a entrega rápida de correções e funcionalidades aceitas. Alcançar a maturidade de entrega contínua não é trivial, principalmente em projetos que envolvem migração de dados ou integração e dependências entre sistemas. A abordagem de implementação de tais práticas deve ser a mesma de qualquer outra tecnologia; um passo por vez, adiciona-se uma nova fase no processo de build e deploy e aguarda-se até a equipe se adaptar à mudança no ritmo de trabalho para adicionar um novo.

Voltar atrás em uma entrega, o processo de *rollback* de um deploy, também é automatizado, utilizando práticas simples, como back-up e restauração, ou, ainda, técnicas de *blue green deployment*[76] para permitir a coexistência temporária em produção de diversas versões de um software. O ambiente do Google App Engine, por exemplo, permite a troca da versão em produção com apenas um clique, dado que ambas as versões já estão implantadas. Esses rollbacks são ainda mais complicados de se implementar, e novamente a equipe deve pesar o benefício da prática contra a necessidade. A experiência indica que, ao maturar as técnicas de entrega contínua, o próximo passo é investir tempo de desenvolvimento técnico no rollback, mas só confiar nele no instante em que for testado em outras máquinas e o efeito al-

cançado for o esperado. O script a seguir é um exemplo simples de suporte a rollback de aplicações Web, baseado no script de deploy mostrado anteriormente.

```
#!/bin/sh

ROLLBACK=$1

if [ -f version ]; then
DUMPVERSION=`cat version`
else
DUMPVERSION=0
fi

DUMPVERSION=$((DUMPVERSION + 1))
echo ‘‘$DUMPVERSION’’ > version

mysqldump -u usuario -p senha sistema \
> snapshots/dump_for_rollback_$DUMPVERSION
cp snapshots/application_$ROLLBACK latest_application.war

mysql -u usuario -p senha -e ‘‘drop database sistema;’’
mysql -u usuario -p senha < snapshots/dump_$DUMPVERSION
```

deploy.sh

Toda essa facilidade de deploy traz um custo para a equipe de infraestrutura e operações, que precisa estar preparada para enfrentar problemas menores, mas possivelmente com mais frequência, no caso da existência de uma quantidade grande de deploys. Deve-se encontrar um balanço entre os dois. Com o passar do tempo, o processo de deploy automatizado contorna muitos dos possíveis erros, minimizando a necessidade de atuação humana ao acontecer alguma falha.

Em aplicações legadas, é comum ver um débito técnico muito grande no processo de deploy, comumente lembrado por sua baixa qualidade e fins de semanas perdidos. A adoção dessas práticas é um caminho longo, que envolve custos, e, portanto, um processo que deve ser adotado aos poucos.

Por fim, adotar longos intervalos entre cada uma das entregas de software leva ao acúmulo de falhas a serem detectadas posteriormente, além de as interpretações erradas dos requisitos reforçarem uma situação cada vez mais difícil de ser contornada.

Não entregar software de maneira contínua é uma grande desvantagem competitiva. Todo o tempo em que uma funcionalidade não está no ar é um ganho às empresas concorrentes em relação ao produto desenvolvido. A Figura 5.4 ilustra a diferença de custos entre empresas que entregam ou não software de maneira contínua:
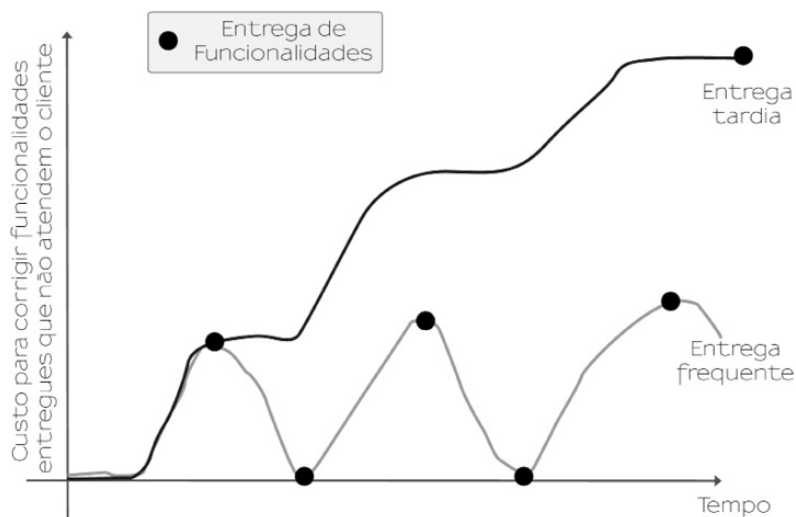
Figura 5.4: Entregar cedo facilita o processo de correção das funcionalidades que não atendem ao cliente.

Diversas boas práticas abordam o processo de criação e manutenção no ambiente dos desenvolvedores, deixando de lado o último trecho a ser caminhado pelo software a fim de ser utilizado pelos clientes: o deploy e a configuração de outros ambientes. As práticas envolvidas com o processo de integração contínua, testes com intenção de capturar erros precocemente, feedback frequente e entrega contínua focam a resolução dos problemas envolvidos nesta última etapa de pré-utilização do software.

Todas essas práticas permitem a criação de testes nos quais duas versões de uma parte do sistema são apresentadas ao cliente (chamadas de *testes a/b*). O teste verifica o retorno de cada variação do sistema para dois grupos distintos de usuários e, após determinado período de tempo, toma-se a decisão de qual das duas ficará em produção, em geral a que produz maior retorno financeiro.

Capítulo 6

Decisões arquiteturais

Como vimos, arquitetura pode ser definida como uma possível interpretação da implementação do sistema. É quando olhamos para a aplicação para observar como diferentes partes do sistema afetam-se entre si. Essa visão mais global traz, porém, diversos desafios e indagações que demandam decisões difíceis que podem significar o sucesso ou o fracasso do projeto.

Quando usar remotabilidade e aplicações distribuídas? Como escolher a melhor abordagem Web entre frameworks *component-based* e *action-based*? Onde usar – e onde não usar – uma tecnologia de mapeamento objeto relacional? Em que momento adotar um arquitetura de comunicação assíncrona? E quais as situações onde usar cloud computing pode ser a melhor ideia?

Esses e outros questionamentos são importantes **decisões arquiteturais** que enfrentamos diariamente nos mais diversos projetos.

6.1 Dividindo em camadas: tiers e layers

Em inglês, duas palavras são comumente usadas ao se falar de divisão de camadas: *tier* e *layer*. Em português, porém, acabamos traduzindo ambas como *camada*, perdendo a importante diferença entre esses termos e dificultando a comunicação.

Ao falar em *layers*, normalmente pensa-se em separações lógicas, em como organizar o código da aplicação de maneira a diminuir o acoplamento e facilitar mudanças. MVC, por exemplo, é um padrão de divisão de *layers* bastante usado. O Domain-Driven Design propõe outra divisão em quatro *layers*, com destaque especial para a *Domain Layer*. Mas há mais divisões de *layers* propostas por outros autores.[67] O importante é promover a diminuição do acoplamento entre diferentes partes do código e evitar que mudanças em um lugar afetem outros.

Já a divisão em *tiers* visa a separações físicas entre partes do sistema. Embora no cenário atual de virtualização e cloud computing seja cada vez mais complicado definir com exatidão barreiras físicas, de um modo geral caracterizamos *tiers* diferentes como componentes do sistema que rodam ou poderiam facilmente rodar em máquinas separadas. O servidor Web é considerado um *tier*, e o banco de dados, outro, assim como o cliente remoto, um servidor de aplicação, um cache distribuído e outros exemplos. Em geral, caracterizam-se como *tiers* diferentes dois componentes da arquitetura que se comuniquem remotamente via rede.

Usar *layers* em excesso pode gerar código de difícil manutenção. E, em relação a *tiers* demais, quando não há uma real necessidade, pode trazer efeitos catastróficos.[90],[158] O tradeoff é claro: adicionar *tiers* pode ajudar em diversos aspectos, mas aumentar a quantidade de invocações remotas, que são caras e complicadas, pode impactar significativamente a performance e a escalabilidade. Fowler diz para reduzirmos ao máximo a distribuição de objetos,[63] e esta é a mesma opinião de outros autores.[13]

Arquiteturas de apenas um *tier* eram comuns antigamente nos mainframes que centralizavam todo o processamento, persistência e interface do programa. Outro exemplo, cada vez mais extinto em aplicações corporativas, são os programas Desktop puramente locais. Certamente, usar apenas um *tier* tem a vantagem de diminuir a latência e melhorar a performance, já que não há a necessidade de efetuar requisições na rede para buscar informações.

Por outro lado, a escalabilidade é comprometida, uma vez que não é possível que um número arbitrário de usuários compartilhe a aplicação. Arquiteturas centralizadas têm também sérios problemas de disponibilidade, já que há um único ponto

simples de falha. Por esses motivos, arquiteturas de dois *tiers* tornaram-se mais comuns.

O exemplo clássico destas arquiteturas são as aplicações Desktop conectadas a um banco de dados central, que se tornaram bastante comuns em aplicações Delphi e Visual Basic (Figura 6.1).
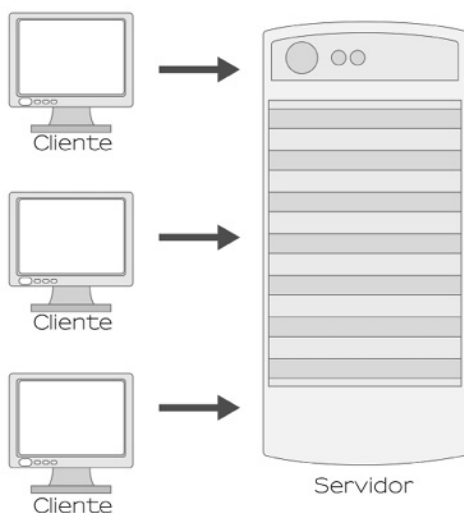
Figura 6.1: Clientes e um servidor.

É muito comum, ainda, o *tier* de interface possuir bastante código relacionado ao domínio. São os *fat clients*, aliviando o servidor de determinadas tarefas que podem ser executadas no cliente. As desvantagens principais nessa abordagem são o processamento mais intenso no cliente e a dificuldade de manter integridade, confiabilidade e segurança. Pensando nisso, uma variação bastante comum é manter a lógica de negócio em *stored procedures* no banco de dados, com o objetivo de deixar a lógica segura e encapsulada, além de minimizar a perda de performance pela adição de mais um *tier*. Usar *stored procedure* talvez seja essencial em uma arquitetura de dois *tiers* por causa de segurança e performance, mas traz os problemas de dependência do banco de dados, dificuldade de integração e pouca separação de responsabilidades (Figura 6.2).

Há ainda o problema do ponto simples de falha no banco de dados central. Mas, embora sua queda possa causar indisponibilidade em todos os clientes, por ser um

fat client, ainda é possível oferecer alguma funcionalidade off-line. Já a escalabilidade nesse tipo de arquitetura está condicionada à capacidade do banco central, ainda mais porque muitas vezes o cliente pode manter uma conexão constantemente aberta. Embora seja uma solução barata, esse tipo de arquitetura tem fortes limitações de escalabilidade e disponibilidade.
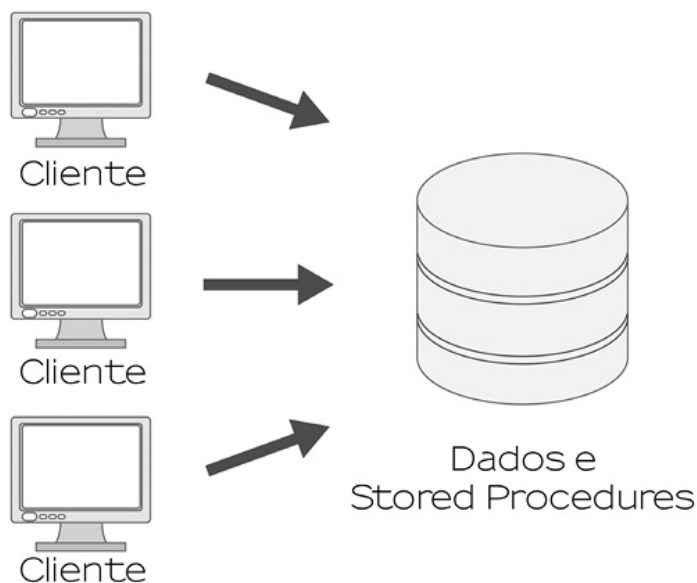
Figura 6.2: Fat client acessando procedimentos mantidos no banco.

Um outro problema enfrentado pelas arquiteturas dois *tiers* é a dificuldade em manter todos os clientes atualizados, prejudicando a extensibilidade, a manutenibilidade e até a segurança. Há tecnologias que minimizam esse problema, como o *Java Web Start*, que fornece uma plataforma simples para a instalação e manutenção das versões de uma mesma aplicação Java desktop. Mas a solução definitiva parece ter vindo com a larga adoção da Web como plataforma para as aplicações.

Na arquitetura Web tradicional, temos três *tiers*: o cliente remoto, o servidor Web e o banco de dados. O cliente é, em geral, um navegador Web que apenas renderiza a interface; não costuma ter processamento e lógica do domínio que não sejam relacionados à visualização dos dados. Esses *thin clients* trazem a imensa vantagem de sempre acessarem a última versão disponível da aplicação, centralizando a evolução do sistema no servidor Web e banco de dados, trazendo maior facilidade de

atualização (Figura 6.3).

Outra vantagem deste tipo de arquitetura é a segurança e confiabilidade, já que o sistema acaba centralizado no *tier* Web, minimizando a possibilidade de clientes maliciosos prejudicarem seu funcionamento. Mas a performance costuma ser um grande problema. Adicionar um novo *tier*, ainda mais Web, acaba aumentando o *response-time*, uma reclamação constante dos usuários que vivenciam uma migração do Desktop para Web.
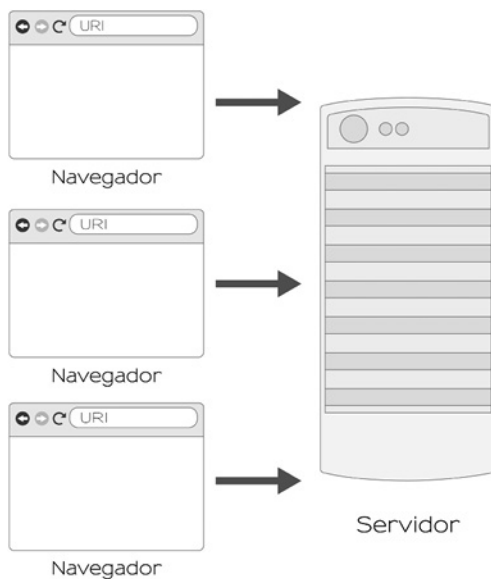
Figura 6.3: Clientes magros através de um navegador e um servidor.

Com o objetivo de melhorar a performance para o usuário, observamos a volta dos *fat clients* na forma de interfaces ricas no navegador, as *Rich Internet Applications* (*RIA*). A própria plataforma Java investe nesse tipo de abordagem desde o início com os Applets e, mais recentemente, com JavaFX, mas podemos citar também Flash, Flex, Silverlight e outros com características semelhantes. Mas essas abordagens são dependentes de tecnologias externas ao navegador e o que mais se vê atualmente é o uso do próprio navegador, para as interfaces ricas, baseadas fortemente em JavaScript, Ajax e até HTML 5. Algumas tecnologias vão além, como o Socket.IO, utilizam WebSockets com fallback para Flash, caso o primeiro não esteja disponível naquele browser. O futuro das aplicações Web é a evolução do navegador para suportar interfaces cada vez mais ricas, mas mantendo as vantagens de uma arquitetura de fácil

gerenciamento.

Ao trabalhar com três *tiers*, deixa de fazer sentido manter todo o processamento no *tier* que armazena os dados, pois o intermediário seria apenas como uma ponte. Nessa abordagem, as antigas *stored procedures* perdem importância e passam a ser substituídas por lógica de negócio executada no *tier* intermediário, o *business tier*. Neste cenário, o cliente faz as requisições ao servidor que executa a lógica e delega a manipulação dos dados para o banco de dados (Figura 6.4).
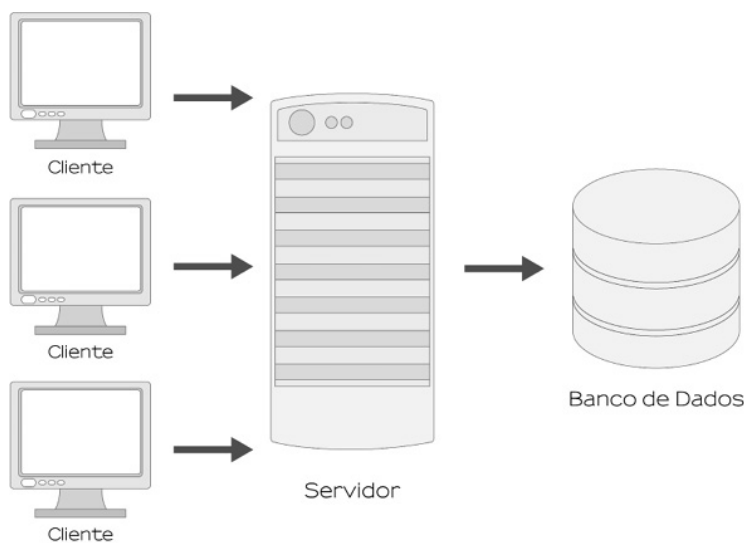
Figura 6.4: Cliente magro acessando processos armazenados no servidor.

Ainda há problemas com disponibilidade e escalabilidade se apenas um servidor Web e um banco de dados estão ativos. Usando literalmente apenas três *tiers* é difícil contornar este tipo de problema, por isso é comum adicionar outros elementos como replicação em *clusters*, *load balancers* e outros *tiers*. É costume chamar este tipo de solução como *N-tiers*, uma variação dos três *tiers* com mais componentes para resolver problemas de escalabilidade, disponibilidade e outros.

Há diversos exemplos de novos *tiers* adicionados às aplicações, como servidores de aplicação separados do servidor Web, caches distribuídos, servidores de mensageria etc. Em geral, há um impacto na performance pela adição de novos *tiers*, mas com melhora significativa em escalabilidade, por haver a possibilidade de se adicionar tantas máquinas quantas forem necessárias, e disponibilidade, por não haver um ponto simples de falha (Figura 6.5).

Este tipo de arquitetura tem também seus pontos negativos, como a complexidade criada e a dificuldade de gerenciamento e manutenção de toda a infraestrutura. Há questões a serem resolvidas no modo de replicação dos servidores se a aplicação tiver estado, além de cuidados particulares com o banco de dados. Por esses motivos, muitos projetos fogem da complexidade do *N-tiers* se não há um requisito muito forte de escalabilidade e disponibilidade.
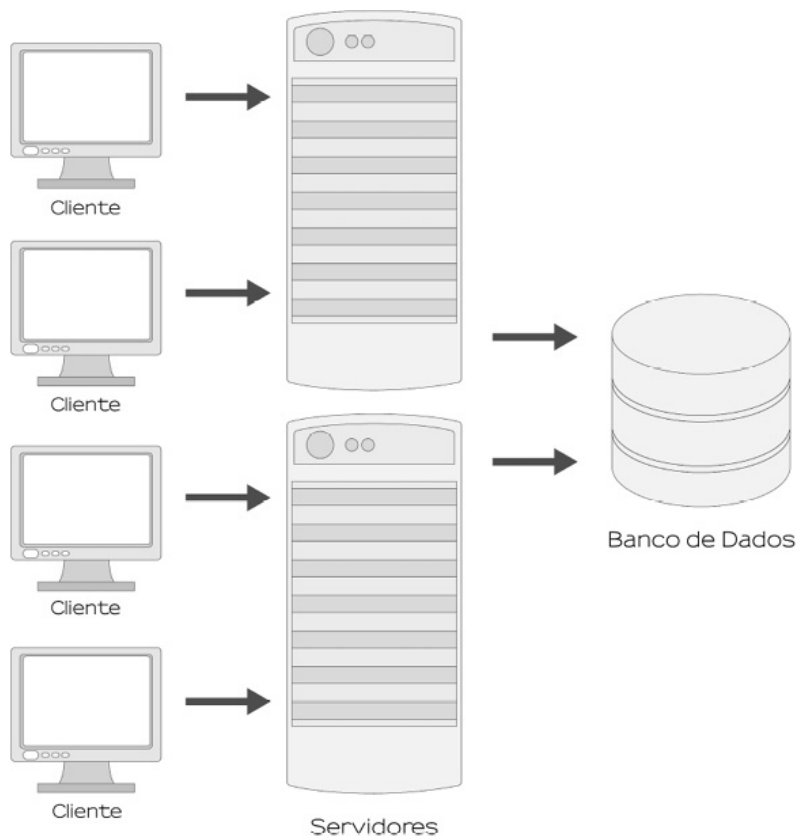
Figura 6.5: Três tiers com diversas máquinas atendendo às requisições.

Otimização na comunicação entre tiers

A Microsoft diz para evitar aumentar o número de *tiers*,[13] assim como a primeira lei de Martin Fowler.[64],[63] Independentemente de em quantos *tiers* sua aplicação está dividida, é necessário tomar precauções. São três os pontos princi-

pais.

O primeiro é a quantidade de invocações para o outro *tier* (número de *roundtrips*). Se o cliente fez uma única requisição, é péssimo gerar diversas requisições internas entre os *tiers* da aplicação. O ideal é que esse número de *roundtrips* seja 1, ou no máximo uma constante pequena. Alguns frameworks podem facilmente gerar uma quantidade proporcional ao número de entidades que estão sendo manipuladas, tornando-se um gargalo enorme para a escalabilidade de aplicação.

O segundo é o tamanho da informação transmitida (*payload*). Quanto maior o JSON, a entidade serializada em XML, mais recursos serão consumidos. Nesse instante, sugem os *Data Transfer Objects (DTO)* para adequar a granularidade do serviço, almejando um único *roundtrip*. O excesso de headers em mensagens curtas também pode ser uma sobrecarga.

O terceiro é o cache, essencial para poder até mesmo evitar uma requisição para o outro *tier*, desonerando muito a aplicação (Figura 6.6).
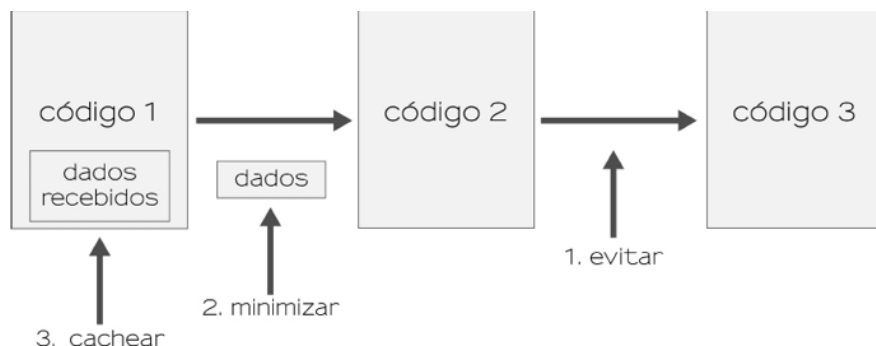
Figura 6.6: Evitar comunicação remota, diminuir os dados transferidos e efetuar cache das informações.

Repare que essas são praticamente as mesmas recomendações dadas por um administrador de banco de dados: minimizar as chamadas, tentando pegar toda informação necessária em uma única query, e ao mesmo tempo buscar apenas os dados necessários. Se possível, cachear o resultado. A comunicação do *tier* de apresentação deve ter os mesmos cuidados; entre os princípios do Google Page Speed para performance Web client-side, estão o cache, minimizar a quantidade de invocações AJAX e compactar os arquivos trafegados para minimizar a quantidade de dados carregados.[189]

Nos próximos tópicos, sempre que um framework envolver trabalho em mais de um *tier*, haverá menção de como tomar esses cuidados na prática.

6.2 Desenvolvimento Web MVC: Actions ou Componentes?

O Java 1.0 surgiu na mesma época do estouro da Web. Mas, como vimos, a primeira abordagem da plataforma era o uso de Applets *client-side*, embora James Gosling já pensasse em algo *server-side* desde 1995. O primeiro movimento em direção à geração de páginas *server-side* foi o lançamento de Servlets em 1997, seguido pelo JSP em 1998.[49]

O grande passo de Servlet foi criar um modelo bastante performático para trabalhar com HTTP em Java sem o desenvolvedor precisar se preocupar com detalhes de infraestrutura ou do protocolo. Com seu processamento de requisições simultâneas em Threads paralelas leves do container, em vez de processos concorrentes, as Servlets se mostraram mais rápidas, escaláveis e robustas que o modelo CGI da época. Isto fez o Java ganhar considerável importância no mercado Web. Mas o objetivo de Servlets nunca foi criar páginas de maneira desacoplada do pensamento dos designers; seu foco era permitir programa usando o protocolo HTTP em Java, com grande performance e escalabilidade. JavaServer Pages nasceu então com o objetivo de trazer produtividade e facilidades na criação das páginas, usando as próprias Servlets por baixo, para aproveitar suas vantagens. No início, o uso de JSP era misturar HTML com chamadas a componentes Java via pequenos pedaços de código Java, os hoje temidos *scriptlets*, desencorajados agora por possibilitar uma péssima mistura de responsabilidades.

Além de ser toda base de qualquer framework e infraestruturas importantes em projetos Web, Servlets são ideais em momentos nos quais precisamos manipular muitos detalhes do protocolo HTTP explicitamente. Páginas dinâmicas geralmente não precisam disto, por isso o uso mais frequente de JSPs em conjunto com frameworks.

No layer de View, vários engines de templating, como *Velocity* e *Freemarker*, surgiram para remover o código Java dos JSP. Mais tarde, muitos dos novos recursos desses frameworks foram incorporados ao JSP 2.0 com uso da *expression language* e *taglibs*, aposentando os *scriptlets*.

A View não precisa ser necessariamente constituída por um template engine como JSP ou Velocity. Se o cliente da nossa aplicação for outro sistema, a View

pode ser implementada através de um processo de serialização de objetos para algum formato, como a do *VRaptor* ao renderizar um objeto em JSON. Nesses casos, uma configuração programática da View substitui um template engine. Em vez de somente páginas *HTML*, este tipo de solução é comum ao disponibilizar serviços ou recursos via Web para consumo por outros sistemas.

A grande evolução no desenvolvimento Web veio com a recomendação do uso do padrão arquitetural *Model View Controller*, o MVC. Os modelos originais, tanto de Servlets quanto de JSPs, não focavam a facilitação da integração com designers, a separação entre lógica de negócios e código *client-side* ou as boas práticas de OO e encapsulamento. Criava-se uma barreira muito forte à manutenção e extensão. O padrão MVC, originário da comunidade Smalltalk,[27] passou a ser aplicado no desenvolvimento Web em Java com a ajuda de vários outros padrões, depois documentados no conhecido livro *Core J2EE Patterns.*[4] A separação entre ***M**odel, **V**iew e **C** ontroller* seria feita, por exemplo, criando uma ca Servlet que fi na frente de todas as requisições, analisando a chamada e delegando para outra classe que realmente saiba realizar a lógica de negócio em questão, fazendo o papel de *Front Controller.*[132]

Logo percebeu-se que as implementações do MVC eram muito próximas, e muito código era independente da aplicação. Surgiram então os primeiros frameworks com implementações reaproveitáveis do padrão MVC, entre eles o **Apache Struts**, um dos controladores pioneiros na plataforma Java.

Além da separação MVC, o Struts ainda trouxe internacionalização, validação e integração com outras tecnologias de mercado. Foi realmente um imenso avanço para a comunidade Java. Com o tempo, porém, o mercado passou a ver deficiências no Struts e a buscar alternativas. Entre os problemas estavam o uso exaustivo de herança de classes do Struts (como a necessidade de estender `ActionForm`), as excessivas configurações em XML e as poucas convenções de código, que geravam um forte acoplamento com o framework.[60] Um dos primeiros concorrentes do Struts foi o WebWork e, depois, vieram outros, como Spring MVC, Tapestry, VRaptor e Stripes. Cientes dos próprios problemas, os desenvolvedores do Struts passaram a investir no **Struts 2** já em 2005, propondo uma junção com o projeto WebWork 2, que possuía um código muito mais flexível.[25]

No meio de toda essa revolução do Struts, nasceu, em 2004, pelo mesmo criador do Struts 1, o **JavaServer Faces**. Foi a primeira especificação Java EE para MVC, fortemente baseada em ideias anteriores de frameworks baseados em componentes, como o ASP.NET, plataforma para desenvolvimento Web criada pela Microsoft. Nesse mesmo ano, surgiu o *Ruby on Rails*, trazendo muitas ideias revolucionárias que

seriam depois absorvidas por vários frameworks. O atraso do lançamento do Struts 2 gerou uma divisão grande no mercado, sem haver a presença de um líder absoluto, como foi com sua primeira versão. Dezenas de frameworks novos apareciam a todo momento, e as novidades do então recente Java 5 estavam ainda começando a ser adotadas. Nessas condições adversas, o mercado viu um porto seguro na nova especificação JSF lançada pelo JCP. Seu modelo de componentes era bastante diferente do que se utilizava em Java até então e, mesmo com sua primeira versão apresentando deficiências e sem os prometidos editores visuais, o JSF atingiu o sucesso com a ajuda da chancela do JCP.

MVC

Mas, independentemente do framework ou da abordagem preferida, o MVC, usualmente tido como um padrão arquitetural, acabou consolidando-se com unanimidade. Há ainda outros padrões e mesmo variações do MVC, mas sempre com o objetivo de facilitar a manutenção e favorecer a separação das *layers*. A Figura 6.7 apresenta uma possível implementação do padrão MVC para uma aplicação Web, conforme sugerido no *Java BluePrints*,[19] e serve como base para a grande parte das soluções que adotam esse padrão.
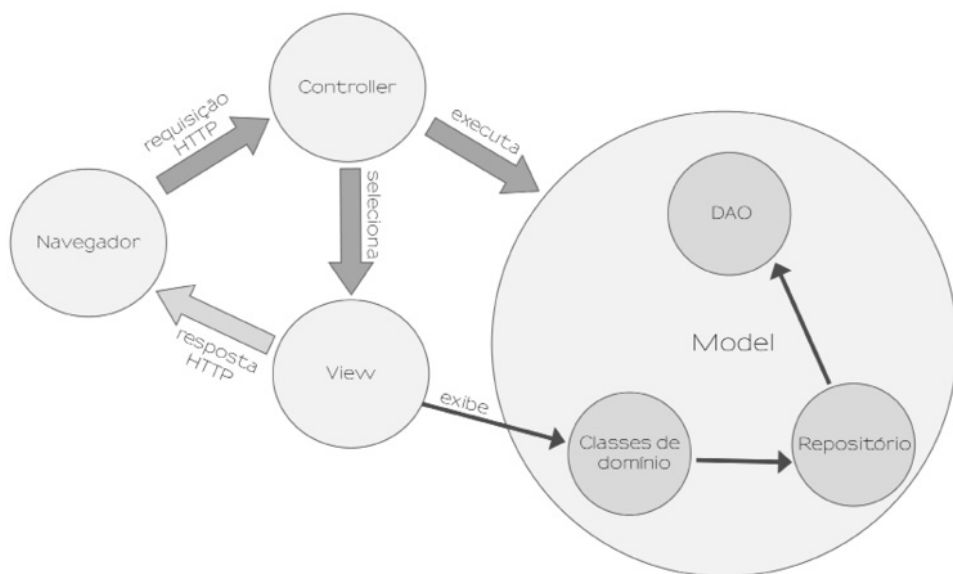
Figura 6.7: Uma possível implementação de Model View Controller para aplicações Web.

O modelo MVC vem justamente para facilitar a separação do layer visual da aplicação do domínio e do negócio. Na definição formal de MVC,[27] a atualização da *view* é feita através de *observers* que são notificados dos eventos ocorridos nos modelos, uma visão muito ligada às aplicações desenvolvidas na época para Desktop. Mas, apesar da definição formal desse padrão ou das interpretações que o mercado adotou do MVC, a característica de maior valor é a de separar os componentes de apresentação do resto da aplicação (conhecido como *Separated Presentation*).[67]

Uma das formas de garantir que isso aconteça é não permitir que um layer acesse outro que não seja seu vizinho direto. Um exemplo que viola esta regra é o acesso direto aos DAOs pelas *Views*, sem passar por alguma Action. Ou, ainda, o acesso a um repositório pela mesma *view* sem passar pelo modelo. Código HTML misturado com SQL e lógica de negócios em Java é especialmente difícil de manter, além de o reúso ser baixo; mas, infelizmente, este ainda é um dos grandes responsáveis pelo alto custo de manutenção de código legado.

Uma grande discussão em torno do modelo é se as Actions (ou equivalente em outros frameworks) pertencem aos modelos ou aos controladores; alguns desenvolvedores acreditam que essas classes são controladores já que, de alguma forma, co-

nhecem a requisição e as *Views*. Os mais puristas, seguindo a definição do padrão, consideram que elas fazem parte do modelo.

Action-based versus Component-based

A maior parte dos frameworks Web existentes segue, naturalmente, o modelo de requisições e respostas do próprio HTTP e também das Servlets. Encaixam-se aqui, entre outros, Struts (1 e 2), Spring MVC, Stripes, VRaptor e frameworks em outras linguagens, como Ruby on Rails, Grails, Django e ASP.NET MVC. Embora todos apresentem diversas facilidades, nenhum tem como objetivo esconder o modelo de requisições e respostas inerente do protocolo HTTP, nem sua característica *stateless*, seu modelo de URIs e tantos outros pontos. Essa categoria de frameworks é comumente referenciada por **Action-based** ou *Request-based*. O exemplo a seguir mostra uma ação que cadastra clientes, escrita através do Struts 2.

```
public class AdicionaClienteAction {
    private Cliente cliente;

    @Action(value=”adiciona” , results = {
    @Result(name = “ok” , location = “/cliente/adicionado.jsp”)
    })
    public String execute() {
        new Clientes().adiciona(cliente);
        return “ok”;
    }

}
```

Em contrapartida, há outra linha de frameworks, liderada pelo JSF e ASP.NET, que traz uma abstração maior no desenvolvimento Web, no qual o objetivo é não observar mais tão de perto os detalhes e o fluxo do HTTP, mas, sim, trabalhar com componentes visuais de alto nível e um modelo de desenvolvimento mais próximo de aplicações Desktop. São os frameworks **Component-based**, nos quais também podemos encaixar Wicket, JBoss Seam, GWT e alguns outros (Figura 6.8).

Figura 6.8: Representando uma tela através de componentes ricos.

Com as incontáveis possibilidades de frameworks Web, mais importante que discutir funcionalidades pontuais de cada um é saber escolher entre *action-based* e *component-based*. Os frameworks baseados em componentes, como o JSF, têm como objetivo trazer características do desenvolvimento Desktop para o desenvolvimento Web. Programa-se orientado a componentes ricos e reaproveitáveis, com um modelo de tratamento de eventos robusto e manutenção do estado das telas (*stateful*). O HTTP, porém, não é feito de componentes, mas orientado a requisições (não eventos), e é *stateless*. Ou seja, um framework *component-based* como o JSF precisa criar abstrações complexas em cima do modelo tradicional da Web.

Alguns desenvolvedores não estão acostumados com o modelo de requisição e resposta da Web, mas, sim, a pensar na integração entre elementos de tela, como como botões ou campos de texto, controlando o que fazer quando um deles dispara um evento, como o clique, mudança de um valor, e outros. A origem do MVC está ligada a esse ambiente Desktop ou console, e não a aplicações Web baseadas em requisições e respostas.[27], [67] Há ainda aplicações Web com interfaces muito complexas, nas quais a interação entre os diversos componentes e seus estados é o ponto principal – não há necessidade de controle fino do HTML gerado, por exemplo. Nesses casos, usar um framework *component-based* como JSF pode facilitar bastante.

Mas há situações em que precisamos controlar o HTML final, como, por exemplo, por causa de acessibilidade, ou para suportar navegadores e plataformas diferentes, ou, ainda, como otimizações para buscas (SEO), e até mesmo controle fino das requisições AJAX, código JavaScript e HTML. Pode haver ainda a necessidade de uma manipulação das URIs utilizadas, tanto para suportar *bookmarks* corretamente quanto para atender aos mecanismos de buscas, e, ainda, permitir redirecionamentos e outros controles. Podemos preferir trabalhar orientados a requisições e respos-

tas com características *stateless* para melhor aproveitar a Web e outras ferramentas que giram em torno do HTTP, além de garantir escalabilidade e disponibilidade mais facilmente. Nesse tipo de situação, usar um framework *action-based* costuma se encaixar melhor como solução.[96]

Porém, é importante ressaltar que é possível usar os dois modelos para resolver praticamente qualquer problema. Não é porque optamos usar *action-based* que não podemos usar componentes visuais ricos, como, por exemplo, com JQuery UI, YUI ou ExtJS. E não é porque optamos por *component-based* que não podemos desenvolver stateless ou controlar melhor as URIs e ter um bom SEO.

Todos os frameworks costumam suportar, a seu modo, recursos como validação, internacionalização, facilidades para popular os *beans* e muitos outros. Em geral, costuma-se dizer que aplicações Web complexas e ricas em formulários, que de alguma forma sejam uma grande aplicação, são mais bem servidas por frameworks *component-based*, enquanto sites Web que exijam um controle mais fino do que está sendo enviado para o cliente são o caso ideal dos *action-based*. Ambos os modelos são capazes de atender a praticamente todos os cenários, e a escolha, em geral, se dá pelo estilo de programação da equipe, que pode ser mais orientado a componentes e eventos ou a páginas e requisições. Aliás, um importante movimento a favor da ideia da coexistência entre ambos os modelos foi o lançamento do ASP.NET MVC pela Microsoft. Durante anos, o ASP.NET clássico foi símbolo da era dos frameworks *component-based*, e a própria plataforma .NET hoje suporta também a abordagem *action-based* com o ASP.NET MVC.

JavaServer Faces

Por ser uma especificação, o JSF teve uma aceitação no mercado muito mais fácil e rápida que muitos frameworks escritos por terceiros. A ideia de trabalhar com uma especificação é muito tentadora para diversas empresas, que precisam de um pouco mais de garantia da continuidade da tecnologia. Mas este não deve ser o único ponto avaliado ao escolher uma tecnologia para determinado problema, muito menos para resolver todos os problemas de uma empresa.

Uma das primeiras propagandas do JSF dizia que seria possível escrever a interface do usuário sem a preocupação com onde ela seria desenhada. Poderia ser em um navegador comum (HTML), em um de celular (WML), ou até mesmo renderizá-la em Flash, sempre com o mesmo código e os mesmos componentes. Bastaria trocar o elemento que renderiza a resposta – chamado *Render-kit* – e teríamos renderizações bem diferentes. Esta característica foi pouco utilizada, pois a necessidade das

empresas era apenas de gerar telas HTML para navegadores comuns; na prática, é raro ver uma aplicação que troque o render-kit padrão do JSF.

Mas essa característica de não escrever a página em HTML puro, e, sim, usando os componentes do JSF, permite que o framework cuide de aspectos que geralmente causam problemas para os desenvolvedores Web, escondendo diversos pontos trabalhosos. Este controle dá muito poder para o JSF fazer, por exemplo, o *binding* nos componentes, permitindo ações que realmente facilitem o desenvolvimento de sistemas Web, como o valor de um campo ser automaticamente guardado em um atributo de um bean já convertido para o formato correto. Ou que o clique de um botão na tela dispare um método escolhido pelo desenvolvedor direto no bean. Ou, ainda, que, ao ocorrer uma mudança de um valor, seja disparada uma validação, entre outros recursos.

O controle que o JSF faz sobre o que acontece nos componentes só é possível porque o framework guarda e manipula uma representação de cada elemento como um objeto, organizando todos eles no formato de uma *árvore de componentes*. Cada elemento da árvore guarda o estado atual dos componentes da tela e, a cada evento disparado, é feita uma requisição para o JSF, que se encarrega de, logo no início da requisição, atualizar as informações nesta árvore.

A árvore de componentes também serviu para amenizar outro problema comum para os novos desenvolvedores Web: o fato de o protocolo HTTP ser stateless. Como entre uma requisição e outra não há armazenamento de informações automaticamente, a não ser que se use sessões explicitamente, o JSF ajudou muito ao guardar informações na árvore, tirando mais um peso das costas do desenvolvedor. Há, por exemplo, um poderoso e complexo ciclo de vida dos componentes e requisições JSF, o que dá ao desenvolvedor a possibilidade de acompanhar as diversas fases do fluxo interno de execução do JSF, como manipulação do estado da tela, validação, conversão, binding, invocação da lógica de negócios, etc.

Além de tudo, o modelo *Component Based* que o JSF usa permite uma grande extensibilidade; é possível criar novos componentes de acordo com a sua necessidade. Algumas empresas até mesmo criaram componentes extremamente úteis e práticos, como, por exemplo, o *RichFaces*, que adiciona à gama oficial de componentes do JSF funcionalidades como componentes mais ricos, ferramentas de layout, temas, entre outros.

O JSF 2 foi uma grande evolução no framework, amenizando diversos antigos problemas e acrescentando importantes novas funcionalidades. Há suporte nativo a Ajax, Facelets integrado, novos escopos, melhor suporte a GET e bookmarks. Além

disso, há diversas convenções que diminuem as configurações que, por sua vez, agora são feitas com anotações simples. Outro ponto bastante forte é a integração com o restante do Java EE 6, em particular as validações simplificadas com Bean Validation, injeção de dependências com CDI e até transformar os managed beans em EJBs para ganhar os serviços do container.

Mesmo com diversas vantagens, o JSF não é unanimidade entre os desenvolvedores. Pelo fato de ser *Component Based*, ainda existem alguns empecilhos para sistemas Web. Entre eles, a falta de controle do HTML é um dos mais sérios. Como o HTML é gerado pelo próprio JSF, qualquer otimização ou modificação que queiramos fazer se torna extremamente custosa e complexa; em casos mais extremos, pode ser que o JSF não dê suporte a determinada funcionalidade. O design e a experiência do usuário são, em geral, adaptados pelas limitações que o JSF apresenta, em vez de uma abordagem mais livre no design, em que o código será adaptado para atingir o resultado desejado.

JSF segue um modelo menos focado na Web e mais voltado para a árvore de componentes. A Web é *apenas* o ambiente do JSF, mas seu objetivo final são os componentes. JSF acaba se tornando uma ferramenta poderosa para criar aplicações com interfaces muito complexas, com diversas funcionalidades, mas que não necessitem de um controle fino do HTML, JavaScript e CSS, ou uma proximidade maior das características do HTTP. Nesses casos, favoreça um framework *Action Based*.

6.3 Domine sua ferramenta de mapeamento objeto relacional

Consultas SQL, gerenciamento de transações, caches e a decisão de quando persistir e buscar dados do banco são preocupações frequentes em grande parte dos projetos e demandam muito tempo e trabalho. Todo código utilizado para resolver esses problemas raramente é escrito de forma a facilitar seu reúso, e o programador é obrigado a reescrever grande parte dele para acessar os dados ao iniciar um novo projeto.

Por este e outros motivos, soluções que visam facilitar o trabalho do programador, diminuindo a necessidade do uso do JDBC e a escrita de complicadas consultas SQL, apareceram há bastante tempo e ganham cada vez mais força.

O que há de errado com o JDBC?

JDBC é uma das APIs mais antigas do Java, existente desde o JDK 1.1 e com constantes atualizações. A versão 4.0, disponível no Java 6, trouxe melhorias significativas

e oferece muito poder até mesmo a bancos de dados específicos.[156] O código para executar uma simples transação com JDBC puro pode ser escrito de maneira bem sucinta.

```
public void insere(String email) throws SQLException {
    Connection con = abreConexao();
    con.setAutoCommit(false);
    con.createStatement().execute(
    ‘‘INSERT INTO Destinatario (email) VALUES (‘’ + email + ‘‘)’’;
    );
    con.commit();
    con.close();
}
```

Pode não estar óbvio, mas este tipo de código ingênuo é a causa dos principais problemas ao acessar bancos de dados com Java: vazamento de memória, transações não finalizadas corretamente, excesso de conexões abertas com o servidor pela ausência de pool, baixa escalabilidade, vazamento de conexões e até mesmo *SQL injection*.

Podemos tentar resolver esses problemas com um código mais completo.

```
public class DestinatarioDao {
    private final DataSource dataSource = new ComboPooledDataSource();

    public void insere (String email) throws DAOException {
        String sql = ‘‘INSERT INTO Destinatario (email) VALUES (?)’’;

        Connection con = null;
        PreparedStatement stmt = null;
        try {
            con = this.dataSource.getConnection();
            con.setAutoCommit(false);
            stmt = con.prepareStatement(sql);
            stmt.setString(1, email);
            stmt.execute();
            con.commit();
        } catch (SQLException e) {
            try {
                if (con != null)
                con.rollback();
                throw new DAOException(‘‘Erro no insert’’, e);
```

```
            } catch (SQLException e) {
                throw new DAOException(‘‘Erro no insert e rollback’’, e);
            }
        } finally {
            try {
                if (stmt != null)
                stmt.close();
            } catch (SQLException e) {
                throw new DAOException(‘‘Erro ao fechar statement’’, e);
            } finally {
                try {
                    if (con != null)
                    con.close();
                } catch (SQLException e) {
                    throw new DAOException(‘‘Erro ao fechar con’’, e);
                }
            }
        }
    }
}
```

O código mais robusto utiliza `PreparedStatement` para evitar SQL injection e tirar proveito de alguns bancos que pré-compilam as queries; toma cuidado com `try`, `catch`, `finally` para não deixar transações pendentes e conexões abertas; usa um pool de conexões através de um `DataSource` para evitar o vazamento destas e não utilizar conexões que possivelmente possam ter sido fechadas pelo servidor.[183]

E este código ainda pode ter melhorias. O esforço em lançar exceptions com boas mensagens e não deixar vazar `SQLException` poderia ser maior com mais blocos `try/catch` para tratar os erros separadamente, como no caso de falhar a abertura da conexão. É válido reparar que a utilização de inversão de controle ajudaria bastante o trabalho de evitar esse código repetitivo (o *boilerplate code*). Mesmo tomando todo este cuidado, há aqui um trabalho ainda de design, já que o programador deve decidir onde colocar o código SQL, isolando-o em DAOs, arquivos de *properties*, XMLs ou, ainda, anotações do JDBC 4.0.

Com a evolução da plataforma Java, surgiram diversas soluções para minimizar o risco e facilitar a implementação dos padrões que aparecem no código anterior.

A primeira opção seria abstrair o acesso direto ao JDBC tomando as devidas precauções e usando as melhores práticas. Ainda assim, teríamos muito trabalho e código repetido, e, por isso, ferramentas conhecidas como *data mappers* surgiram. Es-

sas ferramentas liberam o programador da árdua tarefa de executar a consulta SQL, mapear as linhas da tabela nas entidades do seu modelo, gerenciar transações, etc.

Um framework bastante utilizado é o **iBatis**, considerado por alguns como um *data mapper* menos poderoso, que já encapsula as necessidades comuns do desenvolvedor, quando utiliza o JDBC, e chega até a organizar as queries SQLs em arquivos XML.[14] Apesar de não ser um *data mapper*, o **Spring JDBC template** é mais uma alternativa de uso. Esses tipos de *wrappers*, porém, são pouco usados. Um dos reflexos desta perda de popularidade é que o iBatis parou sua evolução. Um fork foi criado, o **MyBatis**, mas a comunidade Java caminha cada vez mais para as ferramentas de mapeamento objeto relacional.

Mapeamento objeto relacional

O próximo passo seria utilizar uma ferramenta que facilitasse ainda mais o trabalho do desenvolvedor, fazendo com que ele deixasse até mesmo de escrever consultas SQL. Essas ferramentas são conhecidas como **ORM** (*object-relational mapping).* Elas são capazes de fazer a ponte entre o paradigma entidade relacional e o orientado a objetos de forma a minimizar o abismo entre os conceitos dessas duas abordagens, conhecido como a impedância objeto-relacional (*object relational impedance mismatch*).[6]

Um exemplo são as tabelas compridas que aparecem com certa frequência no mundo relacional e, diferente de classes com muitos atributos na orientação a objetos, não são vistas com tanta repulsa. Ferramentas de mapeamento devem possibilitar que mais de uma classe seja utilizada para representar uma única tabela. O contrário também pode acontecer, como uma classe ser mapeada em mais de uma tabela.

Outro caso são as tabelas associativas simples. Elas são a forma de fazer o relacionamento *muitos para muitos* entre entidades no modelo relacional. Ao mapear isso, sua ferramenta deve arrumar uma forma elegante de esconder esses aglomerados de pares de chaves, provavelmente sem criar uma classe para tal e usando apenas coleções e arrays de objetos.

Já a herança é um conceito que não bate com modelagem relacional. Em alguns frameworks ORM, o uso da herança pode formar um sério gargalo de acordo com a forma de representação adotada no banco de dados (as conhecidas estratégias da JPA e do Hibernate), dado que diversas tabelas precisam ser consultadas em algumas queries polimórficas, em especial quando é impossível usar alguns recursos do SQL, como *unions.*[181] Muitos vão desaconselhar a herança, preferindo usar composição,

conforme visto no tópico *Evite herança, favoreça composição*[81], [181] (Figura 6.9).

Com a força que as especificações Java EE têm, é comum que a escolha por uma ferramenta ORM acabe caindo sobre uma das implementações da JPA. Dentre elas, os principais concorrentes são o Oracle TopLink, a OpenJPA, a BEA Kodo, o EclipseLink e o Hibernate, certamente a opção mais comum. Seja qual for sua escolha, há alguns cuidados que devem ser tomados.
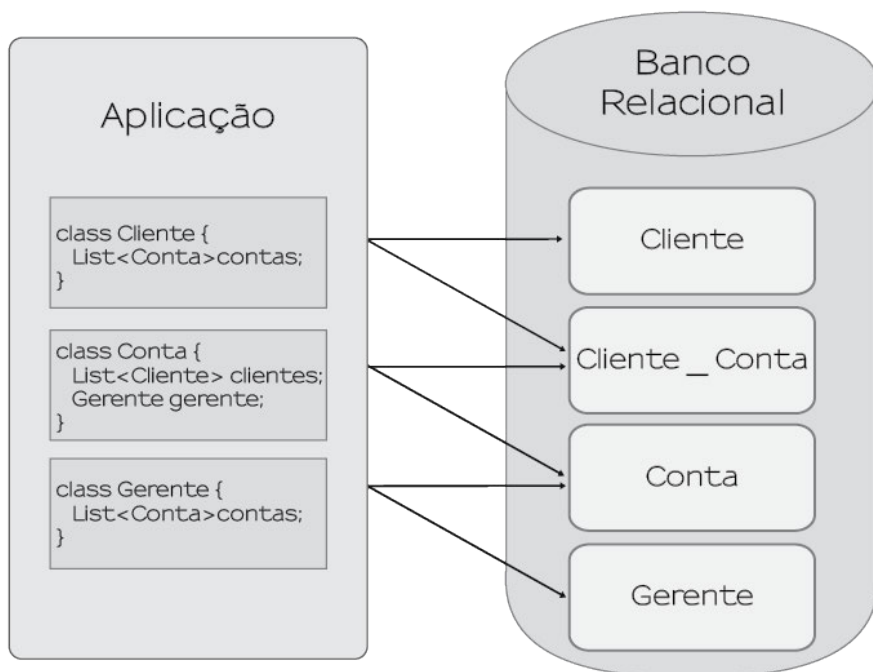
Figura 6.9: Mapeamento de relacionamentos muitos para muitos e muitos para um.

Um cuidado importante é entender e usar corretamente os modos *eager* e *lazy* de se obter objetos relacionados. Sempre que temos relacionamentos com `@ManyToOne` ou `@OneToOne`, o padrão é trazer o objeto relacionado (*eager*); e, quando temos `@OneToMany` ou `@ManyToMany`, a coleção de objetos relacionados só é recuperada quando for acessada (*lazy*). Para boa parte dos casos, o comportamento padrão é suficiente, mas é preciso identificar casos em que precisamos configurar um `FetchType` diferente (por exemplo, quando um objeto relacionado é muito pesado e queremos comportamento *lazy* em vez de *eager*). É preciso estar atento ao uso indiscrimidado de lazy, que pode gerar o problema dos *n+1* selects, como veremos, assim como o

uso indiscriminado de *eager* pode ocasionar carregamento desnecessário de dados em excesso.

Há também a possibilidade de se ajustar o *fetch type* diretamente nas consultas; por exemplo, fazendo uma busca específica como *eager* em que o relacionamento está configurado como *lazy* – usando *join fetch* nas HQL e o `setFetchMode` das Criterias.[155] Algumas implementações de JPA, como a OpenJPA e o Hibernate, ainda permitem que campos básicos sejam *lazy* sem a necessidade de uma pré-compilação, possibilitando que uma entidade seja parcialmente carregada e que campos chaves sejam inicializados somente quando e se necessários.

Outro cuidado a ser tomado é na integração com o restante da aplicação. Na Web, acessamos entidades do nosso modelo na *View* e, se elas possuem relacionamentos *lazy*, precisamos que o `EntityManager` que as carregou esteja aberto até o fim da renderização da página. Caso esteja fechado, a maioria das implementações de JPA não vai se reconectar ao banco de dados e lançará uma exception (`LazyInitializaionException`, no caso do Hibernate).

Para que isso não ocorra, devemos manter o `EntityManager` aberto, seja através de um filtro, de um interceptador ou algum outro mecanismo. Este é o conhecido pattern *Open EntityManager in View* (derivado do *Open Session in View*, termo cunhado pelo Hibernate) (Figura 6.10).
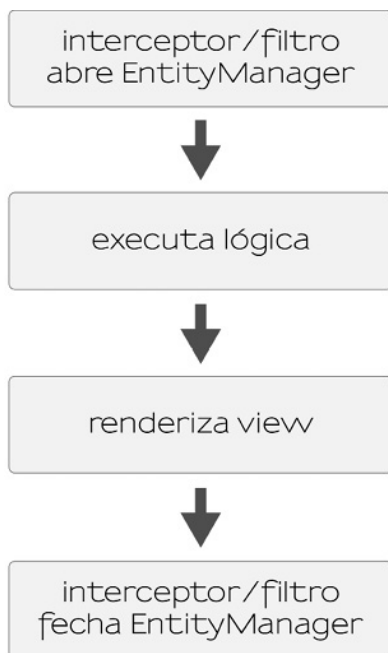
Figura 6.10: Padrão Open EntityManager in View.

Outra forma de escapar dos problemas de proxies *lazy* é utilizar um container de injeção de dependências. Frameworks, como Spring e VRaptor, já possuem filtros e interceptadores para esses casos, e até mesmo usando a anotação `@PersistenceContext` do Java EE para injeção de dependências podemos obter o mesmo efeito.[184]

Com o advento do Java EE 5 e algumas formas de injeção de dependências, diminuiu a necessidade de um grande número de patterns que havia para o J2EE, mas algumas preocupações ainda persistem. A principal delas envolve como acessar um `EntityManager`, uma pergunta para a qual existem diversas opções: dentro de um objeto que age como um DAO, em um Domain Model ou em um Model Façade. Ainda é possível, e cada vez mais fácil com o uso da injeção de variáveis membro, acessá-lo diretamente através da lógica de negócios. Apesar de este ainda ser um debate em que muitas opiniões divergem, a grande maioria dos desenvolvedores opta por proteger o acesso ao `EntityManager` através de um DAO, não permitindo que qualquer código o acesse e também ajudando a reaproveitar métodos que encapsulam rotinas de acesso a dados que são mais que apenas a execução de uma

query.[153],[185]

Por uma questão de ajuste de granularidade, para que não sejam expostos mais dados que o necessário, ou para agrupar diferentes atributos de entidades em um único resultado, é possível que haja a necessidade do *Data Transfer Object*.[154] Outros patterns do Java EE 5 também foram catalogados, mas muitos deles não foram largamente adotados pela comunidade, ou possuem nomenclatura e implementações divergentes.[138],[154]

Connection Pool

A quase onipresença do Hibernate, tanto através do uso da JPA quanto diretamente, faz com que o conhecimento profundo das funcionalidades oferecidas por este framework mereça atenção especial. Apesar do esforço da JPA 1.0 em unificar uma grande parte de funcionalidades comuns a todas as ferramentas ORM em uma especificação, muito ficou de fora. A JPA 2.0 tenta diminuir essas grandes diferenças entre os diversos fabricantes existentes, criando novas anotações e estendendo a interface `EntityManager` com novos métodos e recursos.[46]

Entre os recursos importantes que existem no Hibernate e outros frameworks, porém não na JPA, podemos citar alguns que são vitais para a escalabilidade e a performance da aplicação. O desconhecimento dessas funcionalidades pode levar até mesmo um projeto simples ao fracasso, seja pelo consumo excessivo de memória, seja pelo disparo exagerado de queries.[182]

Usar um *pool de conexões*, por exemplo, é praticamente obrigatório em qualquer aplicação que use banco de dados. Um problema comum em aplicações que abrem conexões indiscriminadamente é sobrecarregar o banco com um número muito grande de conexões. O uso do pool permite controlar o máximo de conexões que podem ser abertas. Além disso, abrir e fechar conexões são operações custosas. Não vale a pena estabelecer toda a comunicação com o banco, abrir a conexão, para usá-la rapidamente, e depois logo fechá-la. Usar o pool economiza bastante tempo ao manter as conexões abertas e compartilhá-las.

Há diversos pools de conexão no mercado já implementados e prontos para uso, desde os próprios *data sources* dos servidores até bibliotecas separadas como o C3P0 e o DBCP. Usar um pool de conexões com o Hibernate é extremamente fácil, bastando umas poucas linhas de configuração XML,[93] lembrando-se apenas de evitar o pool padrão do framework, que deve ser utilizado apenas para testes.

Um problema tradicional dos pools está ligado à detecção de conexões quebradas, que ainda estão marcadas como disponíveis, causando a famosa

`java.net.SocketException: Broken pipe` ao tentar utilizá-la. É necessário checar de alguma forma se a conexão de um pool ainda está válida, aberta, seja através de uma verificação, ao requisitar uma conexão, seja executando verificações periódicas.[183]

Cache e otimizando a performance

Outro ponto importante para uma boa performance e escalabilidade é evitar consultas repetidas em pequenos intervalos de tempo: o sistema responde rápido e o banco de dados não é acionado. Dentro de uma mesma `Session`, as entidades gerenciadas por ela ficam em um cache, o de primeiro nível (*first level cache*), até que essa sessão seja fechada ou a entidade explicitamente removida (através do `clear` ou `evict`). Esse primeiro nível de cache existe para garantir a integridade dos dados na memória e no banco, evitando que no contexto de uma `Session` possa existir mais de uma representação do mesmo registro ao mesmo tempo. É o que Fowler chama de *Identity Map* em seu POEAA.[67]

Pensando em um contexto Web, o cache de primeiro nível existe durante o ciclo de vida de uma requisição. É possível ir além, já que muitas entidades sofrem poucas alterações e podem ter seus dados cacheados por uma região maior que a delimitada por uma única requisição. O **cache de segundo nível** (*second level cache*) tem este papel. Através de simples anotações, é possível tirar proveito de robustos mecanismos de cache, como *Ehcache* ou *JBoss Cache*. É possível ainda configurar detalhes como a política do ciclo de vida das entidades no cache (LRU, LFU, FIFO...), seu respectivo tamanho, os tempos de expiração, entre outros.[212]

Quando uma determinada entidade for buscada através de sua chave primária e ainda não estiver no cache de primeiro nível, o Hibernate passará pelo cache de segundo nível para verificar sua presença. Caso seu tempo de expiração não tenha passado, uma cópia desse objeto cacheado será devolvida sem necessidade de acesso ao banco de dados. Se as tabelas em questão são apenas acessadas por sua aplicação, é possível configurar o cache de tal maneira que nunca expire enquanto não houver uma atualização na entidade. O código a seguir mostra a utilização transparente de um cache de segundo de nível, uma vez que a segunda busca não atingirá o banco de dados:

```
EntityManager manager1 = jpa.getEntityManager();
Conta primeiroRetorno = manager1.find(Conta.class, 15);

EntityManager manager2 = jpa.getEntityManager();
```

```
Conta segundoRetorno = manager2.find(Conta.class, 15);

System.out.println("Titular da primeira conta: " +
primeiroRetorno.getTitular());

System.out.println("Titular da segunda conta: " +
segundoRetorno.getTitular());
```

Na Figura 6.11 é possível verificar como os dados são armazenados e encontrados no cache de primeiro e segundo níveis:

Fazer um cache eficiente, que evita vazamento de memória e problemas de concorrência sem perder escalabilidade, é uma tarefa difícil; ao adotar ferramentas de ORM que suportam tais bibliotecas, toda essa infraestrutura já está disponível. Mas toda estratégia de cache é perigosa porque, para evitar ficar com dados obsoletos, é preciso uma boa política de invalidação em caso de atualizações. O Hibernate faz tudo isso de forma transparente se as atualizações são feitas através dele. Mas, se nossos objetos tiverem muitas atualizações em comparação com o número de consultas, o efeito pode ser o inverso do desejado: o Hibernate gastará muito tempo nas invalidações de cache e a performance provavelmente será prejudicada. **Use cache apenas em entidades que não mudam nunca ou nas que mudam muito pouco em relação às consultas**.
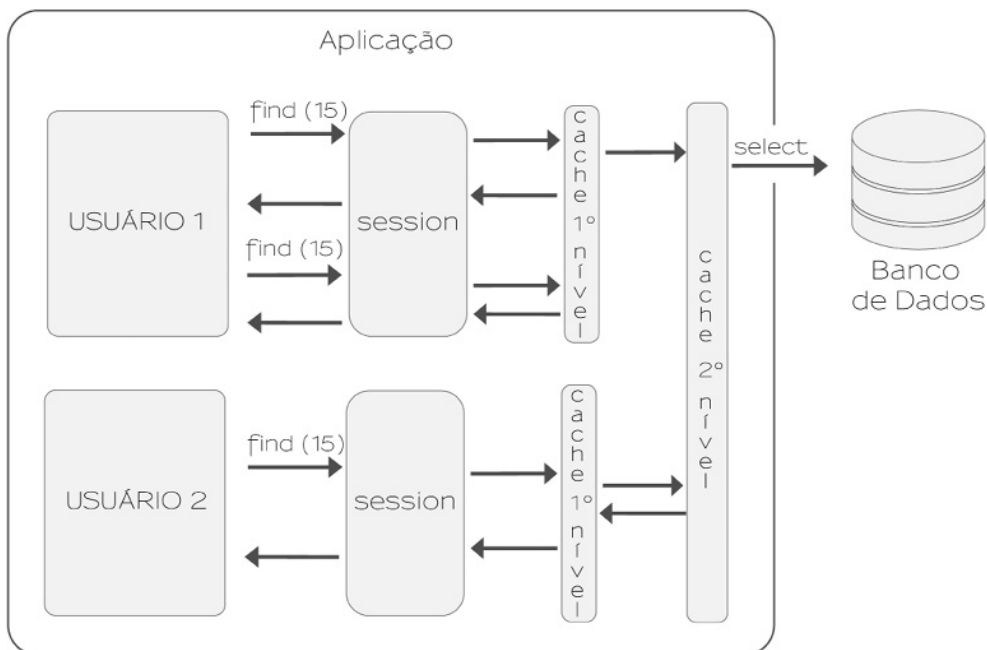
Figura 6.11: Cache de primeiro e segundo níveis em ação.

A adoção de caches favorece performance e escalabilidade em troca de maior consumo de memória. Caches são ainda mais eficazes para aplicações nas quais dados desatualizados (*stale*) podem ser mostrados por determinado período de tempo sem prejuízo para o cliente. Por exemplo, uma query que retorna o produto mais vendido no dia pode ser cacheada por alguns minutos ou até horas.

O que foi apresentado até agora é o chamado cache de entidades. Mas frequentemente são executadas pesquisas que retornam o mesmo resultado até que as tabelas envolvidas sofram alterações. O Hibernate pode cachear até mesmo o resultado de queries, através do *query cache*. Guardar todo o resultado de uma query consumiria muita memória e dados repetidos, e é por isso que a ferramenta vai apenas armazenar as chaves primárias das entidades devolvidas. Quando esta for executada mais uma vez, o Hibernate pegará esse conjunto de chaves primárias e as buscará, sem a necessidade de executar a query de novo. Por isso, é importante também colocar essas entidades no cache de segundo nível, e não apenas a query, para que a nova execução não precise fazer nenhum hit no banco de dados, utilizando apenas os dados da memória para devolver o resultado.

Esse resultado de query cacheado será invalidado ao menor sinal de modificação nas tabelas envolvidas, pois isso pode alterar o que esta query retornaria. Queries que envolvam tabelas modificadas constantemente não são boas candidatas ao *query cache*.

É possível que o desenvolvedor não queira invalidar todo o cache de uma query a cada modificação nas respectivas tabelas, pois diversas vezes é aceitável que a resposta para uma query não seja consistente com os dados mais atuais, ou seja sabido que a atualização de determinados campos não influenciam o resultado das queries. Esse comportamento de invalidação do cache de queries pode ser modificado para um ajuste mais fino, através da substituição da `StandardQueryCache`, implementando diretamente a `QueryCache` ou estendendo o padrão, para que seja tolerante a alguns updates.[95]

Em queries muito pesadas e frequentemente acessadas, uma implementação simples, que verifica a existência da informação no cache e então executa a query, pode possuir gargalos. Se, por exemplo, no intervalo de 10 segundos, 200 usuários acessarem essa informação e ainda não havia dados no cache, a query será executada uma vez para cada um deles, já que o resultado só é colocado no cache após o retorno da primeira requisição. Esse problema, quando muitas requisições se acumulam em cima de um dado, que pode ou não estar cacheado, é chamado de *dog pile effect*.[97] O *double check locking* e outras boas práticas previnem situações como essas, e é importante conhecê-las.[171],[18],[83] A integração dos frameworks com as bibliotecas de cache já se previnem dessas situações, porém, ao utilizar a biblioteca de cache diretamente, e em alguns casos isto se faz necessário, você mesmo precisa tomar essas medidas.

Quando o volume de dados é muito grande, uma máquina pode ser pouco para guardar todo o cache. É possível utilizar um cluster para armazená-lo, e o Hibernate já trabalha com *Infinispan*, *Coherence* e *Memcache* de forma a diminuir a necessidade de configuração. Com eles, é possível fazer o *fine tuning* para minimizar o tráfego intracluster, além de desabilitar a sincronização imediata de estado com o banco de dados, usando o *write-behind*, uma abordagem assíncrona muito mais escalável, porém com menor confiabilidade, pois permite a visualização de dados desatualizados (*stale data*).[208],[216]

Mais otimizações e recursos

Além das estratégias de cache, o Hibernate ainda disponibiliza outros mecanismos de otimização. Ao executar uma query, é comum iterar sobre todas as entidades

retornadas. Se o resultado de uma query é um List<Livro>, e Livro possui como atributo uma List<Autor> que será carregada de maneira *lazy*, existe então um potencial problema: ao fazer o laço para mostrar todos os autores desses livros, para cada invocação livro.getAutores() uma nova query será executada, por exemplo:

```
select * from Autor where livro_id = 1
select * from Autor where livro_id = 3
select * from Autor where livro_id = 7
select * from Autor where livro_id = 12
select * from Autor where livro_id = 15
select * from Autor where livro_id = 16
```

Este problema é conhecido como n+1, pois, ao executar uma única query, tem-se como efeito colateral a execução de mais n queries, onde n é o número de entidades resultante da query original e que pode ser bem grande.

É possível configurar o Hibernate para que ele inicialize a coleção *lazy* de autores dos livros de uma maneira mais agressiva, sem fazer cada carregamento separamente, mas sim de 5 em 5, valor que pode ser definido através da anotação @BatchSize(size=5), que diminui bastante as queries de inicialização:

```
select * from Autor where livro_id in {1, 3, 7, 12, 15}
select * from Autor where livro_id in {16}
```

Assim, o número de queries executadas é minimizado, unindo diversas queries em uma única, diminuindo o número de *roundtrips* até o banco de dados.

Para detectar a presença do problema n+1 e outros, não é viável olhar o tempo todo o log de queries em produção, pois é fácil deixar escapar diversos gargalos. O Hibernate possui a classe Statistics que, se habilitada, armazenará dados sobre entidades, relacionamentos, queries e cache. É possível ver com detalhes quantas vezes cada query está sendo executada, os tempos máximos, mínimos e médios; e, ainda, quantas vezes o cache de segundo nível é invalidado em comparação a quantas vezes ele é usado; ou até quantas transações foram abertas e não comitadas, entre outros. Através desta classe, é muito mais fácil identificar gargalos e problemas para sabermos onde colocar um novo cache, onde usar o @BatchSize, onde é melhor ser *eager* ou *lazy*, etc.

Ainda falando em otimizações, o Hibernate é muitas vezes criticado em situações que envolvem o processamento de muitos dados de uma vez. Mas existem mecanismos específicos para trabalhar nesses casos.[92] Imagine que seja necessário fazer o processamento de um imenso arquivo TXT e inserir milhões de linhas no banco

de dados de uma vez. A invocação de `session.save` (ou `entityManager.persist`) várias vezes, provavelmente, vai resultar em uma `OutOfMemoryError`.

A noção de contexto persistente atrelado à sessão faz com que todos os objetos inseridos sejam armazenados no cache de primeiro nível até que ela seja fechada. E inserir muitos objetos nesse cache não é uma boa ideia. Soluções possíveis vão desde chamar `session.clear` de vez em quando para limpar esse cache, até usar diretamente a `StatelessSession`, outro tipo de sessão que não guarda estado (e por isso perde vários benefícios do Hibernate, mas resolve problemas de manipulação de muitos objetos).

Já quando o problema for trazer muitos registros do banco para a memória, melhor que executar uma simples *query* e pegar uma imensa lista de objetos que provavelmente estourará a memória, é estudar estratégias de como pegar poucos objetos por vez. Nesse cenário, os `ScrollableResults` permitem a utilização do cursor diretamente no banco de dados, evitando trazer todos os dados de uma vez para a memória.

Deve-se considerar até se vale a pena trazer os elementos para a memória. Muitos processamentos podem ser feitos diretamente no banco de dados através de *batch updates*. Por exemplo, reajustar em 10% os preços de todos os produtos de uma loja: em vez de trazer todos os produtos para a memória e atualizar chamando o `setPreco`, podemos executar um único *UPDATE* diretamente no banco de dados. O Hibernate suporta esse tipo de processamento através de *UPDATE*, *DELETE* e *INSERT... SELECT* diretamente pela HQL.

É também possível executar queries nativas pelo Hibernate. Embora uma das grandes facilidades do framework seja a geração das queries, existem situações nas quais queremos escrever algumas delas diretamente. Um bom motivo é aproveitar as queries complexas, escritas pelo DBA, com todos os detalhes de otimização do banco, em vez de tentar traduzir tudo para o HQL. Com queries nativas, conseguimos chamar recursos bem específicos do banco de dados que estamos usando, inclusive *stored procedures*, e obter um grau a mais de otimização nas situações em que performance for mais importante que portabilidade de banco de dados.

Há ainda diversas outras funcionalidades, como a existência do modo de *fetch extra lazy*, fazendo com que uma coleção não seja inteiramente inicializada: queries diferentes serão disparadas para chamadas, como `lista.size()`, `lista.get(15)` e até mesmo `lista.add (entidade)`, evitando carregamento de dados desnecessários. Ou, ainda, as funcionalidades de auditoria do Hibernate, que permitem manter um histórico completo de alterações em cada entidade do sistema.

Além de funcionalidades próprias, o Hiberante possui diversos subprojetos que auxiliam tarefas comumente relacionadas ao banco de dados. O **Hibernate Search**, por exemplo, integra de maneira elegante o Lucene com o Hibernate, permitindo que seu índice de pequisa textual seja atualizado de maneira transparente através de listeners, e possui mecanismos de alta escalabilidade para fazer a atualização destes de maneira assíncrona. A busca textual muitas vezes é deixada de lado por desconhecimento do Lucene e desta integração, gerando queries complexas que geram resultados pobres, já que o SQL não permite buscas com o mesmo poder.

Não usar algumas dessas poderosas funcionalidades pode dificultar o desenvolvimento da aplicação, além de possivelmente criar um enorme gargalo de performance. É importante conhecer a fundo as funcionalidades que afetam a performance do Hibernate,[94] além de todos os outros recursos para aumentar a produtividade do programador. Novamente aqui, **usar um framework sem um conhecimento significativo sobre suas funcionalidades pode causar problemas que seriam facilmente evitados.**

6.4 Distribuição de objetos

Distribuir objetos em diferentes máquinas é uma proposta comum para tentar melhorar a escalabilidade de uma aplicação. Mas as dificuldades e desvantagens que uma arquitetura distribuída apresenta podem ser mais prejudiciais à performance e manutenibilidade do que os ganhos.

Essas vantagens e desvantagens da distribuição dos objetos dependem do design adotado pela implementação, que pode induzir os desenvolvedores a cometer erros com facilidade. Historicamente, por escolhas inadequadas no design de diversas tecnologias de objetos distribuídos, criaram-se design patterns para resolver os problemas inerentes à remotabilidade, como os relacionados à perda de escalabilidade ao fazer invocações remotas em excesso sem necessidade, ou ao transportar um volume alto de dados entre os *tiers*.

Com cada versão nova do EJB, por exemplo, as invocações remotas ficam mais transparentes e se aproximam demais a uma invocação local, quando olhamos apenas o código. O desenvolvedor sem muito conhecimento pode acabar codificando e gerando inúmeras requisições entre tiers, causando perda de performance e escalabilidade, problemas comuns ao empregar tais soluções sem o devido cuidado.

Martin Fowler, conhecido por seus trabalhos em torno de boas práticas de OO e patterns arquiteturais, formulou a *Primeira Lei de Objetos Distribuídos*: "**Não dis-**

tribua seus objetos".[64],[63]

Ao criar interfaces de granularidade grossa nos *Facades*, ao copiar objetos em DTOs e em muitas outras situações impostas pela remotabilidade, sacrificam-se as boas práticas de design OO, diminuindo a flexibilidade e a extensibilidade. E este preço é alto demais para se pagar na maioria dos casos. De modo geral, Fowler prega que devemos favorecer um sistema bem projetado e flexível a uma arquitetura distribuída. É também pelo motivo da complexidade e do custo de invocações remotas que a Microsoft recomenda o uso de arquiteturas não distribuídas quando possível.[13] Programar pensando em Facades com métodos de granularidade maior é muito diferente do que se está acostumado a fazer com uma única JVM.

Os principais argumentos para uso de arquiteturas distribuídas com EJBs são escalabilidade e disponibilidade. Uma divisão muito comum para isso é separar o *Web server* do *Application server* em dois *tiers* (Figura 6.12).
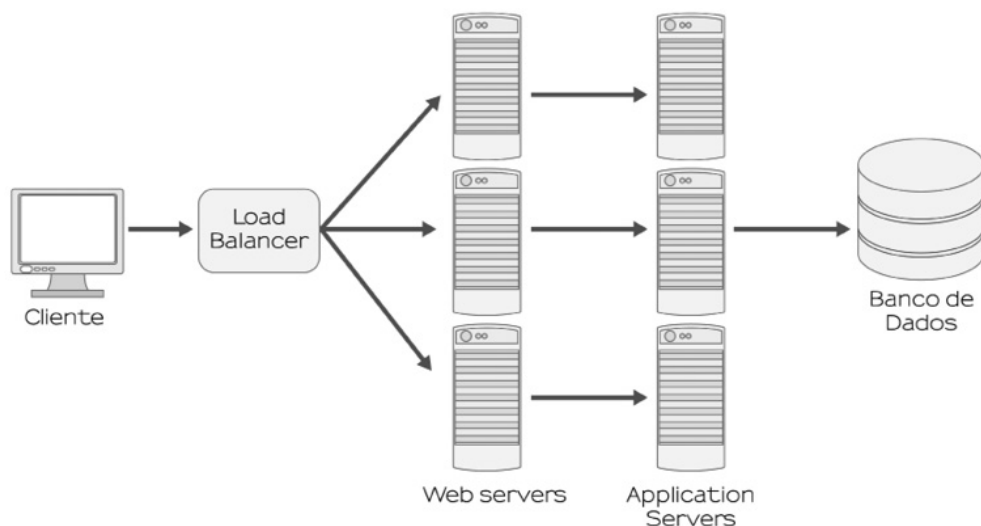
Figura 6.12: Divisão entre Web server e Application server em uma arquitetura N-tiers.

Mas é possível ganhar escalabilidade e disponibilidade sem essa separação; podemos criar toda a aplicação e depois rodá-la em um cluster, onde cada máquina terá uma instância completa da aplicação. Evitando manter o estado na máquina, fica fácil obter redundância num cluster assim, sem necessidade de replicação e sincronização de informações entre seus nós.

Há casos, claro, em que existe a necessidade de distribuir objetos. Esta não deve ser a regra, mas a exceção. Programadores devem ser críticos e parcimoniosos na distribuição de objetos, e Fowler chega a brincar que "*você deve vender sua avó predileta antes de distribuir seus objetos*".[67] Mesmo tentando evitar todas as situações de distribuição de objetos, muitas vezes você não conseguirá eliminar todas.

EJBs

É comum que discussões sobre objetos distribuídos envolvendo a plataforma Java toquem no assunto EJBs. Desde seu nascimento dentro da IBM, esta tecnologia foi fortemente divulgada como parte fundamental da arquitetura Java EE, e que teria papel importantíssimo em todo tipo de projeto Java. Muitos projetos passaram a utilizar EJBs por causa disso, muitas vezes sem refletir o suficiente sobre essa escolha. E logo surgiu o grupo dos críticos ferrenhos aos EJBs, questionando as vantagens e apontando as desvantagens de adotar a tecnologia.

Antes mesmo dos EJBs, com o objetivo de expor a interface de um objeto para acesso remoto, foi criada a especificação do RMI, uma tecnologia que só provê o recurso de remotabilidade. Os *Enterprise JavaBeans* foram inicialmente desenvolvidos pela IBM, em 1997, como um container de serviços em cima do RMI. Incorporados em 1998 pela Sun à plataforma Java, o EJB tinha como objetivo prover um *container* que disponibilizasse implementações de diversos aspectos e necessidades comuns às aplicações corporativas.

Entre esses serviços, temos persistência, controle transacional, segurança, mensageria, *Web Services*, serviço de nomes, pooling de objetos e muitos outros. Como é uma especificação do Java EE, tudo isso está pronto e implementado nos diversos servidores de aplicação disponíveis para uso em qualquer aplicação Java. Com a evolução da plataforma, esses recursos ficaram mais fáceis de ser utilizados pelas aplicações, além de aumentar a quantidade de recursos padronizados, diminuindo as funcionalidades específicas de container. Conhecer bem o que cada um desses serviços traz é importante para tomar a decisão de adotar ou não o modelo de componentes do EJB.

Os *Stateful Session Beans* são os que mais se assemelham à ideia de um objeto exposto pelo RMI, mas com serviços adicionais. É um objeto no servidor que guarda o estado da sessão com o cliente. Cada cliente possui sua própria instância, onde pode armazenar seus dados e chamar métodos remotos de negócio (Figura 6.13).
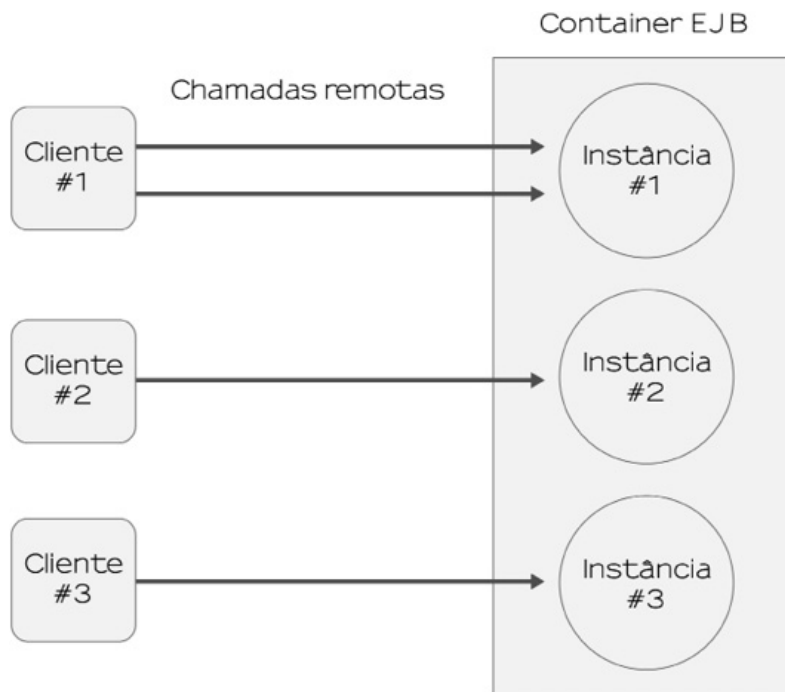
Figura 6.13: Stateful Session Beans com instâncias reservadas para cada cliente.

Para atingir alta escalabilidade – uma das vantagens do EJB –, o container possui um mecanismo de passivação e ativação de objetos. Como o número de instâncias pode ser muito grande em aplicações com muitos usuários simultâneos, o container é capaz de temporariamente serializar alguns dos objetos da memória que estejam sem uso no momento (*passivação*). E, tão logo o usuário necessite novamente da sua instância, o container vai reativar na memória de modo transparente (*ativação*), sem necessidade de código extra nenhum por parte dos desenvolvedores da aplicação.

Esse tipo de comportamento, somado à capacidade de replicação e clusterização dos servidores de aplicação, é o que faz os EJBs serem tão usados para atingir alta escalabilidade e disponibilidade.

Já os *Stateless Session Beans* tentam tratar de outra categoria de uso, na qual se quer a distribuição do objeto e os serviços do container, mas onde não é necessário guardar informações relativas a um cliente. É um objeto que expõe métodos de lógica de negócio sem necessidade de ser utilizado sempre pelo mesmo cliente (Figura 6.14).
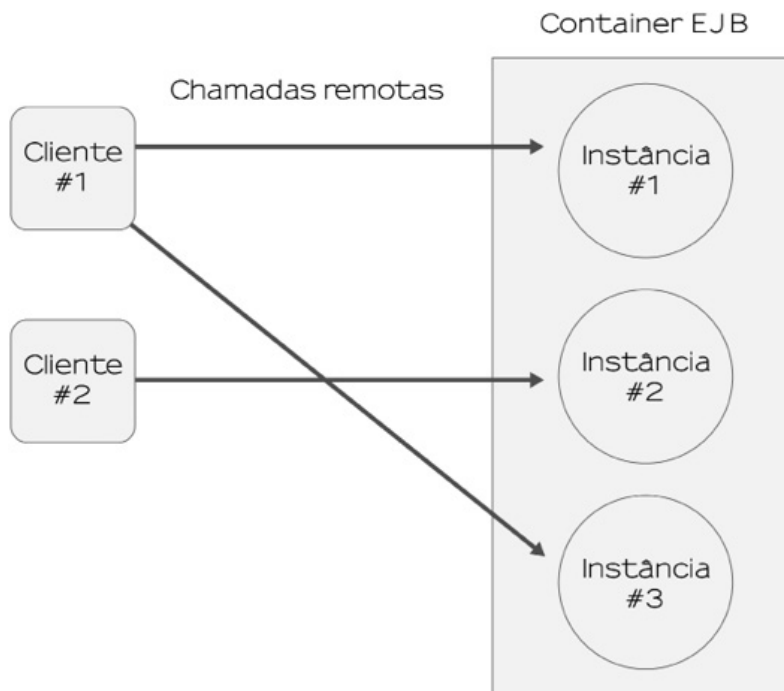
Figura 6.14: Pool de instâncias do Stateless Session Beans compartilhadas por todos os clientes.

Mas o nome desse tipo de EJB é extremamente confuso. O termo *stateless* sugere que o objeto não pode ter atributos e manter estado. Mas ele pode ter estado, desde que não seja vinculado a algum cliente específico. Dizemos que ele não pode é manter *estado conversacional*, ou seja, toda conversa com o cliente deve ser feita em uma única invocação de método; se ele invocar um segundo método, será uma nova conversa, e possivelmente até com um objeto diferente.

É por este motivo que o container mantém um *pool* com várias instâncias compartilhadas entre os diversos clientes. Os clientes não sabem qual instância acessarão do pool a cada nova invocação, por isso não devem manter estado relacionado ao cliente. É muito comum usar este tipo de EJB para fazer um pool de conexões, por exemplo, onde cada objeto mantém uma conexão aberta em um atributo seu. Para o cliente, todos os objetos são equivalentes, sem estado que o atrele àquela instância específica.

Portanto, a palavra *stateless* pode confundir, assim como a palavra *session*, que

poderia indicar que o objeto guarda sessão (estado conversacional), justamente o que ele não faz. Os *stateless session bean* costumam ser usados apenas para expor métodos de negócio independentes entre si, que executam uma funcionalidade e devolvem alguma resposta sem esperar necessariamente por uma próxima invocação do mesmo cliente. Ele pode até manter o estado de sua instância, mas não um estado conversacional com o cliente.

Os *Entity Beans* fecham os tipos de EJBs que estavam disponíveis desde as primeiras versões da plataforma. Mas, ao contrário dos *session beans*, a partir da versão 3.0, eles deixaram de ser objetos remotos, e, da versão 3.1 em diante, sua especificação, a Java Persistence API, é até mesmo uma outra JSR. O objetivo dos *entity beans* era oferecer um serviço de persistência no container EJB. Seriam objetos gerenciados pelo container, representando entidades do banco de dados sem que o desenvolvedor precisasse se preocupar com toda a persistência. É um serviço bastante útil, mas eles eram remotos.

Imagine um sistema de uma loja virtual que faça uma busca e retorne uma lista de `Produto`. Se precisarmos percorrer essa lista de, digamos, 100 elementos, invocando `getNome` e `getPreco` em cada um, ao usarmos *entity beans*, teríamos 200 invocações remotas de métodos, que é um gargalo gigante para a escalabilidade do sistema pelo excesso de comunicação entre tiers. Justamente para resolver este problema, foi que o *Core J2EE patterns* passou a pregar o uso do *Data Transfer Object* o DTO (antigamente chamado de *Value Object* por esse catálogo) (Figura 6.15).
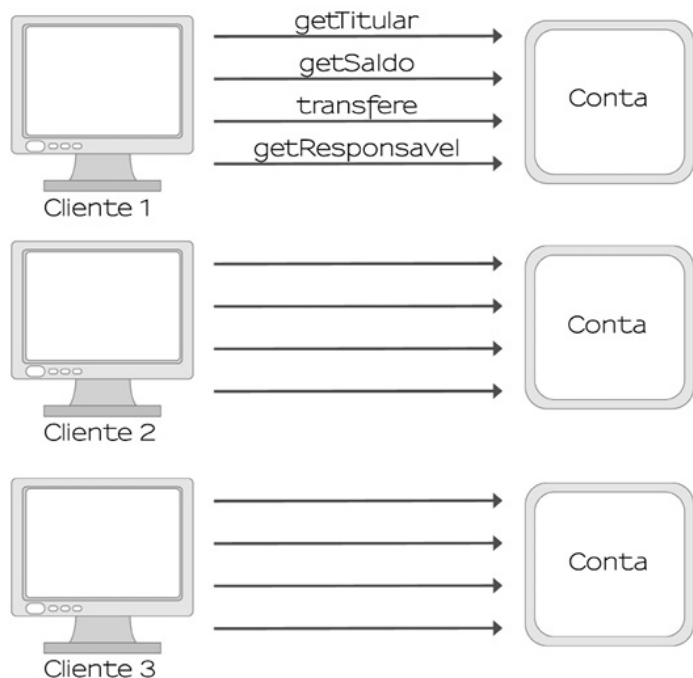
Figura 6.15: Clientes invocando métodos remotos em uma frequência muito alta.

Em vez de devolver uma referência remota do *entity bean*, o padrão nos mostra que os dados devem ser copiados para outros objetos simples, não remotos e serializáveis. Desta forma, o cliente conseguiria os objetos desejados para uso localmente dos dados, sem o preço do grande número de invocações remotas. O padrão acabou virando um remendo da especificação de EJB; quando era usado entity bean, sempre precisávamos aplicar o DTO. Esta prática ficou tão forte, que o DTO começou a ser aplicado em outras situações, muitas vezes sem necessidade; por exemplo, para transportar dados entre *layers* em vez de *tiers*.[67],[154] Alguns até mesmo aplicavam DTOs no MVC para trafegar objetos entre o Model e a View, copiando e colando atributos e adicionando complexidade sem necessidade.[28]

Com o EJB 3.0, o problema da necessidade obrigatória do DTO passou a ser resolvido pela Java Persistence API, onde as entidades não são mais remotas, e sim serializáveis; o pattern foi absorvido pelo framework. Para diminuir a quantidade de requisições ao banco de dados, pode-se utilizar a construção `select new Classe` da JPA-QL, que permite ser mais específico com os dados a serem carregados, através do uso de Value Objects de granularidade mais ajustada, que, para alguns autores, vão

agir como um DTO neste caso.[154] Há também situações novas para se lidar, como potenciais problemas com relacionamentos *lazy* não inicializados antes da serialização. E ainda outros antigos patterns, além do DTO, que também perdem espaço, como *service locator* para buscar dependências, substituído por injeção de dependências e CDI.

Outro tipo de EJB, que entrou na versão 2.0, são os *Message-drivenBeans*, que permitem o processamento de mensagens assíncronas com JMS. É diferente dos demais EJBs por não possuir interface remota, pois é apenas um componente dentro do servidor, usufruindo da infraestrutura de serviços do container e possibilitando maior escalabilidade. Vamos tratar de mensageria e JMS no próximo tópico deste capítulo.

O último tipo de EJB acrescentado à plataforma foi o **Singleton Bean**, novo na versão 3.1. É um objeto global com estado compartilhado por toda a aplicação, garantindo a unicidade mesmo em ambiente clusterizado. É útil para expor dados e comportamentos que sejam necessários em vários pontos da aplicação de maneira segura em relação à concorrência e com todo o suporte do container.

Remoto ou local?

Quando os EJBs surgiram, como vimos, foram apresentados como uma evolução do RMI. Era uma tecnologia para facilitar o uso de objetos distribuídos e ir além, manipulando também seu ciclo de vida, segurança, persistência e outros serviços de infraestrutura.

Uma importante evolução na versão 2.0 foi o suporte a interfaces locais. A partir desta versão, um EJB poderia ter, além da interface remota, uma local que permitiria que o objeto fosse usado eficientemente por outros objetos na mesma JVM e não fosse exposto remotamente. As interfaces locais foram essenciais para se implementar alguns dos conhecidos patterns do Java EE em relação à remotabilidade, como o DTO. Nesse cenário, por exemplo, o *entity bean* é acessado pela sua interface local no servidor, e nunca é diretamente exposto remotamente, evitando invocações remotas em excesso (Figura 6.16).
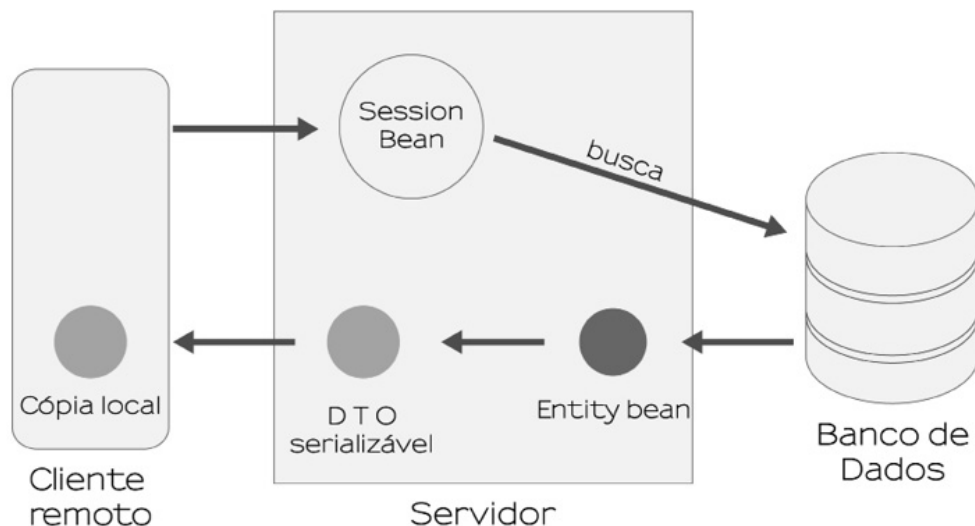
Figura 6.16: Padrão Data Transfer Object.

Outro padrão muito utilizado era o *Session Facade*,[139] também do Core J2EE Patterns. Em um sistema bem construído de acordo com as boas práticas de OO, encapsulamento e separação de responsabilidades, criamos uma série de pequenos objetos, cada um com sua responsabilidade distinta, e todos colaborando para executar as lógicas do sistema. Mas o problema é que, em um sistema distribuído como os EJBs, expor ao cliente muitos objetos com pequenos métodos acaba implicando uma granularidade muito fina. Isto é, para executar alguma operação, o cliente acabaria efetuando muitas invocações remotas para acessar vários métodos e objetos, causando problemas sérios de escalabilidade. O *Session Facade* propõe a criação de novos objetos remotos com interface de granularidade mais grossa para serem expostos remotamente ao cliente, para que ele invoque apenas esse Facade remotamente, e o Facade, por sua vez, faz a invocação aos diversos objetos de negócio envolvidos naquela operação usando suas interfaces locais. Esse pattern ainda é muito usado no EJB 3, apesar de seu nome não ser mais citado com a mesma frequência. É essencial em qualquer arquitetura distribuída preocupar-se com a granularidade dos serviços disponibilizados (Figura 6.17).
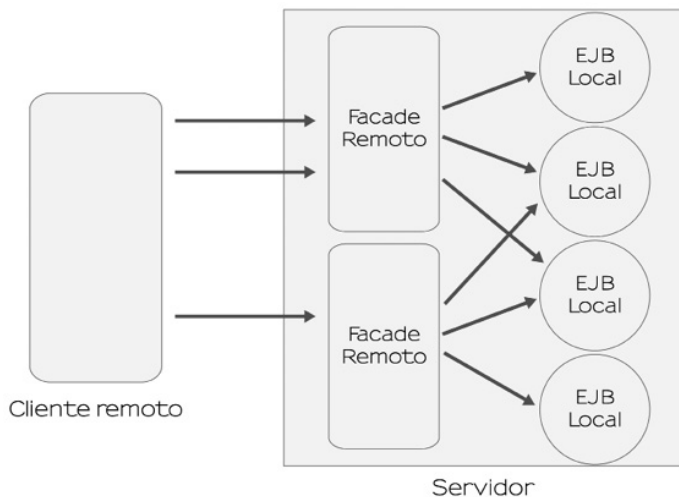
Figura 6.17: Usando Facades para diminuir a granularidade em chamadas remotas.

É bastante interessante acompanhar a história dos EJBs com relação à distribuição de objetos. Lembrando que, originalmente, era uma tecnologia estritamente distribuída, mesmo que a invocação ocorresse em uma mesma máquina virtual. Na versão 2, as interfaces locais foram incluídas para ajudar a implementar os patterns que resolvem os problemas de escalabilidade, evitando o excesso de invocações remotas. Nessa época do EJB 2 é que críticos da plataforma, como o próprio Martin Fowler e Rod Johnson, passaram a pensar em alternativas como POJOs, containers leves e o Spring, pois em muitos casos a remotabilidade e a distribuição de objetos não eram necessárias.

Os EJBs foram muito utilizados como uma alternativa mais fácil ao CORBA, em especial para o caso mais simples de possibilitar o acesso de diferentes clientes, na qual as aplicações Web, mobile e Desktop acessem a funcionalidade em comum. As novas versões do EJB não focaram esta abordagem, dando mais ênfase aos outros aspectos de infraestrutura. Hoje, uma opção frequentemente adotada é expor essas funcionalidades como serviços Web, como será visto no Capítulo 7.

O JCP capturou cada vez mais as tendências dos mercados, aprimorando-as dentro das JSRs; tanto que o EJB 3.0 foi uma grande evolução na plataforma, trazendo simplicidade e produtividade nunca antes vistas no Java EE, boa parte fruto dessas críticas anteriormente apresentadas. A versão 3.1 trouxe ainda uma pequena modificação de imensa importância para toda essa discussão: para usar EJBs localmente,

não precisamos mais nem de interfaces locais. Basta uma anotação e o bean pode ser usado em qualquer lugar da aplicação, usufruindo dos serviços do container EJB, sem a complexidade histórica da plataforma. E, com os novos recursos de inversão de controle e injeção de dependências do CDI no Java EE 6, usar esses beans localmente com todos os serviços é muito simples.

O primeiro a advogar a favor de um container leve de serviços para objetos locais foi o Spring, sempre muito crítico dos EJBs. Com o Java EE 6, finalmente a plataforma ganha uma padronização para um container leve de objetos locais com EJB 3.1 e CDI, resolvendo muitas das antigas críticas. O Spring, por outro lado, acabou acrescentando remotabilidade, Web Services e outros recursos à sua plataforma. Por caminhos e direções diferentes, Java EE e Spring se aproximaram muito em suas últimas versões. Há quem diga que EJBs hoje são mais produtivos com suas anotações, e há quem prefira os recursos a mais do Spring, como AOP e integração com pequenas outras bibliotecas.

Mas fato é que, se você busca um container leve de objetos com serviços reaproveitáveis e uma excelente implementação de IoC e DI, hoje é possível escolher entre Java EE 6 e Spring. EJB deixou de ser sinônimo de remotabilidade.

6.5 Comunicação assíncrona

Existem outras maneiras de distribuir comportamento entre diversas partes de um sistema buscando escalabilidade e maior performance. Em uma abordagem **assíncrona** de comunicação entre sistemas, a troca de mensagens é feita de tal maneira que o sistema que envia a mensagem não precisa esperar o processamento da resposta, ficando livre para continuar sua rotina, diferente de tecnologias de integração com RPC. É uma invocação não blocante.

A característica assíncrona, a principal dos *middlewares* orientados a mensagem (*MOM*), é uma ótima opção para sistemas que necessitem executar rapidamente algum tipo de rotina ao ser disparado um evento. Por exemplo, ao mudar o preço de um produto quando uma promoção é lançada, não é preciso esperar todos os preços serem alterados; o usuário pode continuar utilizando o sistema normalmente. Situações em que não há necessidade de resposta de um comportamento a ser executado são passíveis de implementação via mensageria assíncrona.

A ideia de trocar mensagens entre sistemas remotos não é tão recente assim. Desde os anos 1960, a IBM já possuía o "*IBM 7740 communication control system*", que trabalhava com o fluxo de mensagens entre sistemas remotos.[103] Mas a figura

real do MOM apenas se tornou popular nas décadas de 1980 e 1990, por causa dos produtos como *TIBCO Rendevous* e *IBM MQSeries*, que já forneciam implementações completas de MOM.

Porém, não havia uma padronização que cuidasse da troca de mensagens. Os fabricantes deviam dar suporte a todo tipo de método e bibliotecas de envio e recebimento de mensagens existentes. Foi neste cenário que surgiu a especificação JMS. Como praticamente todos os servidores de mensageria possuíam interfaces de comunicação parecidas, assim como os bancos de dados, foi possível criar uma padronização da biblioteca de comunicação com os servidores de gerenciamento de mensagem.

JMS – *Java Message Service*, em conjunto com diversas outras especificações e frameworks, foi um dos responsáveis pela aceitação do Java como plataforma de desenvolvimento corporativo. JMS carrega consigo as grandes vantagens de um Middleware Orientado a Mensagem: garantia de entrega, controle da entrega, ser assíncrono e escalável, tudo isso na forma de uma especificação, com suporte à integração nativa com a plataforma Java EE.

Após o lançamento do JMS, muitos fabricantes de MOM implementaram a especificação, facilitando o trabalho das empresas no trabalho com mensageria. Sendo uma especificação, não importa a implementação, o modo de enviar e receber mensagem é o mesmo. A Sun foi apenas a líder da especificação; quem mais participou da criação do JMS foram os próprios fabricantes, porque necessitavam de uma padronização.[164]

A estrutura planejada no JMS usa o fato de a troca de mensagem ser assíncrona para garantir a entrega da mensagem, usando o modelo *store-and-forward*. Quando uma mensagem é enviada ao MOM, primeiro ele persiste essa mensagem, depois a repassa para seu destino. Mesmo que o destino não esteja respondendo, a mensagem já está guardada. Então, quando ele voltar a responder, o MOM realiza a entrega. Esse mecanismo é ótimo para trabalhar com serviços que podem não estar disponíveis 100% do tempo. Essa confiança também pode ser configurada; em alguns casos pode não ser crucial entregar todas as mensagens, e algumas falhas de entrega são aceitáveis em troca de uma escalabilidade muito maior.

Como existe um *broker* (intermediário) de mensagens, os sistemas acabam não se comunicando diretamente; quem envia a mensagem não precisa saber nada sobre quem a recebe. Este desacoplamento é muito usado para ligar sistemas que não se conheciam previamente; eles apenas precisam passar a conhecer o *broker* de mensagem, e não um ao outro, oferecendo modularização e flexibilidade para os sistemas.

Essas duas características são extremamente importantes para garantir o reúso de componentes.

O fato de as mensagens serem assíncronas e o sistema *store-and-forward* fazem com que as mensagens sejam processadas apenas na medida do possível, e aquelas que ainda não foram processadas ficam guardadas esperando uma oportunidade. Mesmo que haja um aumento do volume das mensagens, os enviadores não sofrem nenhum impedimento para mandar outras mensagens, e quem as recebe não perde nenhuma delas graças ao *broker*. Ainda que o componente que receba as mensagens não as esteja processando em um tempo aceitável, o desacoplamento entre quem envia e quem recebe permite acrescentar mais processadores de mensagem, ou até mesmo trocar o *broker* por um mais performático.[165]

Modelos de Entrega

JMS disponibiliza dois modelos de entrega mais comuns, o *point-to-point* e o *publish-subscriber*. O *publish-subscriber*, representado pela interface `Topic` do JMS, usa o modelo um-para-muitos para enviar as mensagens. Cada publicação de uma nova mensagem fará com que o MOM confira todos os que estão registrados para receber este tipo de mensagem, e cada um recebe uma cópia, à medida que o *broker* achar conveniente. Esse modelo é recomendado para mensagens que possam ser processadas por zero, um ou vários recebedores. Caso não haja ninguém registrado no MOM para receber a mensagem, ela será perdida.[166]

Pequenas mudanças podem ser feitas nesse modelo, obtendo-se resultados bem diferentes. Ele pode trabalhar com garantia de entrega ao configurar um *subscriber* como um *durable subscriber*; neste caso, o servidor reterá a mensagem até o *durable subscriber* voltar a responder. Outra configuração muito utilizada é o filtro de mensagens, no qual é possível configurar o assunto da mensagem e cada *subscriber* aplicar um filtro.

Já o *point-to-point* usa o modelo um-para-um para enviar as mensagens, e é representado pela interface `Queue` do JMS. Neste modelo, o produtor envia e o MOM entrega para apenas um consumidor. Ele também pode ser usado para balancear a carga de processamento; quando uma mensagem chega ao *MOM*, ele escolhe apenas um consumidor para processar a mensagem, mesmo que tenhamos mais consumidores resgitrados.[167]

A especificação JMS apenas define a interface de comunicação entre os clientes e o servidor JMS, porém, o formato interno da mensagem pode ser escolhido livremente pelas aplicações. Utilizar um formato texto tem prevalecido, mesmo que todos

os participantes sejam Java, pois evita-se utilizar objetos serializados como conteúdo das mensagens. Mensagens XML também são encontradas, apesar de formatos orientados a coluna serem mais abundantes e, de acordo com algumas restrições no volume de dados, até mesmo preferíveis.

A ideia de troca assíncrona não exclui a necessidade de resposta, há caso em que ela apenas não precisa ser imediata. Por isso, a especificação JMS possui recursos para configurar para onde a resposta da mensagem deve ser enviada. Essa opção é muito utilizada: em vez de o consumidor responder diretamente para o produtor da mensagem, ele encaminha a reposta para outra fila, criando um fluxo de operações.

Um exemplo seria o sistema de disparos de e-mails em massa, no qual, em um caso comum, a lista contém milhares de destinatários. Se a requisição for esperar o fim do envio para prosseguir, ela ficará bloqueada por muito tempo. Realizar esta tarefa de maneira assíncrona, encaixa-se perfeitamente, em especial se não houver a necessidade de saber quando as mensagens foram consumidas e se os e-mails foram devidamente enviados.

As vantagens estão diretamente ligadas à restrição; o produtor está livre para operar da maneira que achar conveniente após o envio, não precisando aguardar uma resposta para prosseguir. Por um lado, temos o acoplamento menor entre quem são os consumidores e quem são os produtores, uma vez que as mensagens não são endereçadas para máquinas específicas, mas a classe e o conteúdo da mensagem são fatores de um acoplamento entre as duas partes.

Caso os consumidores assumam um padrão fixo de mensagem dada uma classe específica, qualquer mudança no formato dessa mensagem por um produtor (mesmo a adição de uma informação nova) quebrará os consumidores. Isso mostra o alto acoplamento devido ao formato da mensagem. Se a implementação dos consumidores ignora informações desconhecidas, adições na mensagem não quebram consumidores desatualizados; é a abordagem *Must-Ignore*.

As mensagens assíncronas não aparecem apenas no *business tier*. O Ajax tira proveito do assincronismo para dar uma usabilidade melhor à interface Web, e até mesmo abordagens assíncronas existem para realizar queries no banco de dados, como interfaces análogas ao JDBC, tendo callbacks para receber o resultado quando necessário.

6.6 Arquitetura contemporânea e o Cloud

Boa parte dos sistemas atuais usa alguma variação da arquitetura *N-tiers*, que vimos no tópico 6.1. A clássica arquitetura cliente-servidor *2-tiers* apresentava diversos problemas, entre eles a dificuldade na escalabilidade e uma disponibilidade ruim por causa da dependência ao servidor central. A evolução para arquiteturas Web em *3-tiers* solucionou diversas questões, como gerenciabilidade, extensibilidade e manutenibilidade. Mas sistemas em *3-tiers* ainda têm bastante dificuldade com escalabilidade e disponibilidade.

A evolução para arquiteturas de *N-tiers* permite, assim, clusterização, load balancers, modularização em sistemas, etc. O principal objetivo deste tipo de arquitetura é atacar os problemas de escalabilidade e disponibilidade com suas diversas práticas e tecnologias. EJBs, por exemplo, podem ter características negativas em certos cenários, como um impacto em performance, mas é inegável que melhoram escalabilidade e disponibilidade quando bem utilizados. Ou, ainda, o uso de mensageria e JMS, uma técnica com objetivos fortes em melhorar a escalabilidade e disponibilidade de sistemas integrados. Muitas vezes, preferimos escalabilidade horizontal, em vez de vertical, apenas pela melhora de disponibilidade e escalabilidade praticamente infinita.

Disponibilidade costuma ser um requisito importante para a maioria dos sistemas, até os pequenos. Escalabilidade é algo mais exigido por sistemas maiores, mas mesmo pequenas aplicações estão sujeitas a picos momentâneos de acesso ou a um crescimento imprevisível de usuários. Porém, embora importantes e vastamente discutidos, escalabilidade e disponibilidade costumam ser grandes dificuldades para os arquitetos. Mesmo com as diversas ferramentas disponíveis e servidores cada vez melhores, é preciso se preocupar em como montar o cluster Web de Jettys. Ou, ainda, de que forma ativar a replicação do banco de dados. E como escolher os pontos da aplicação que necessitam de caches distribuídos? Qual deve ser sua infraestrutura física para que ela escale o suficiente e quais as características do hardware escolhido (*dimensioning*)?

São muitas decisões e uma arquitetura bastante complexa. E a pior parte, talvez, é ficar na dúvida se a aplicação vai realmente aguentar. Se o concorrente quebrar e houver uma migração em massa de clientes para a empresa, o sistema escala na velocidade necessária? Se houver um problema com o datacenter, a aplicação estará disponível em outra instalação? Se for necessário um grande volume de acessos momentâneos, consigo atender à necessidade pontual sem desperdício depois? Se acontecer um grande desastre natural, tenho replicação em outra cidade? Outro

continente?

Resolver essas questões é possível e existem diversas técnicas para planejar a capacidade necessária, mas todas bastante trabalhosas, caras e com uma possibilidade de erro em casos não previstos. O arquiteto acaba, então, deixando de lado problemas que julga de menor risco para o sistema em questão. E se fosse possível, porém, ter alta disponibilidade e escalabilidade com baixo custo, sem complexidade e com menos planejamento?

Em 2007, o jornal americano *The New York Times* enfrentou um desses problemas.[86] O novo sistema *TimeMachine* permitiria aos leitores o acesso ao acervo histórico do jornal, um total de mais de 11 milhões de artigos em edições publicadas ao longo de 150 anos. Todo esse material é armazenado em terabytes de imagens TIFF de alta definição, com cada imagem representando pequenos pedaços de uma página impressa.

O trabalho consistia em gerar PDFs amigáveis dessas páginas em uma resolução razoável para consumo via Web. Mas quantas máquinas e quanto tempo seriam necessários para gerar PDFs para 150 anos de história do jornal? E, mesmo que se chegasse a um número "x" necessário, qual seria a melhor estratégia: comprar diversas máquinas para esse uso pontual e desperdiçá-las ao fim da tarefa? Ou usar o tempo ocioso de máquinas já existentes para evitar desperdício, mas prolongando bastante o tempo de conclusão da tarefa? A solução no *New York Times* foi usar **Cloud Computing**. Mais especificamente, resolveram o problema em 24h com 100 instâncias rodando na Amazon EC2 ao custo irrisório de algumas poucas centenas de dólares.

Cloud computing é uma revolução no mundo de TI, tanto que é o foco do Java EE 7. Seu grande atrativo é a facilidade para escalar todo tipo de tarefa com baixo custo e sem preocupação com manutenção da própria infraestrutura. Há até mesmo quem compare essas mudanças com a revolução elétrica do começo do século XX.[32] A invenção da energia elétrica transformou o mundo no século XIX, mas durante muito tempo esteve restrita a empresas e instituições que podiam adquirir seus próprios geradores. A revolução no século XX foi o surgimento de grandes usinas geradoras e de uma extensa rede de distribuição. A produção em alta escala derrubou os preços da energia e o serviço de distribuição simplificou seu uso. A eletricidade logo se popularizou até para famílias de mais baixa renda, já que o serviço era barato, simples e sem complexas infraestruturas próprias. E, sendo um serviço, paga-se apenas pelo que se usa, tornando fácil o controle de orçamento e dispensando altos investimentos em infraestrutura.

Cloud computing é a computação como serviço. Como os serviços de energia elétrica, água encanada ou telefonia, o cloud encara os recursos computacionais como serviços. Processamento, armazenamento, tráfego de dados e outros passam a ser oferecidos pelos provedores de cloud para uso simples e barato. Em vez de manter a própria infraestrutura de servidores, temos à disposição serviços que oferecem recursos computacionais de acordo com cada necessidade. Não é preciso um grande investimento inicial, e as preocupações com manutenção e gerenciamento da infraestrutura praticamente desaparecem. É possível escalar quase que infinitamente, de maneira elástica, usando recursos somente quando for necessário e liberando-os quando não forem. Evita-se desperdício ao se pagar apenas pelo que realmente for usado. No fim, é uma grande economia de custos e dores de cabeça.

Diversas grandes empresas proveem serviços de cloud computing, como Google, Microsoft, Amazon, SalesForce, Rackspace e Locaweb. Mas há diferenças entre os serviços oferecidos. Alguns provedores, mais notadamente a Amazon, com seu EC2, oferecem a virtualização de hardware onde é possível rodar máquinas virtuais completas. Chamados de **IaaS** (*Infrastructure as a Service*), esses serviços oferecem grande flexibilidade para todo tipo de aplicação, mas ainda exigem a gerência de toda a camada de software, assim como a comunicação entre todos os nós. Isto pode ser facilitado e automatizado através de ferramentas, mas está sempre sob responsabilidade do desenvolvedor.

Pensando em mais facilidade, diversos provedores disponibilizam plataformas completas de desenvolvimento, abstraindo não só o hardware, como todo o gerenciamento de sistema operacional, servidor Web e softwares básicos. Esses serviços, chamados **PaaS** (*Platform as a Service*), são oferecidos, por exemplo, pelo Google App Engine, o VMForce, Amazon BeanStalk, Heroku e Microsoft Azure. Desenvolve-se a aplicação, usando as ferramentas disponibilizadas pelo provedor, e o único trabalho é o deploy na nuvem. Não há preocupação com o gerenciamento de hardware nem do software de infraestrutura, permitindo que se foque apenas na aplicação. Com isso, há bastante facilidade para desenvolver e subir aplicações, com escalabilidade elástica, alta disponibilidade e pagando apenas pelos recursos usados.

Ter o ambiente todo pronto é a grande vantagem do PaaS, mas pode ser também sua desvantagem. Muitas vezes, é difícil, ou até impossível, ter um controle fino das configurações e softwares usados na plataforma. No App Engine, por exemplo, não se controla o número de instâncias da aplicação, é preciso confiar nos algoritmos de escalabilidade da plataforma. Há também uma série de restrições em relação a quais APIs Java podem ser utilizadas, além de limitações ao sistema de arquivos, banco de

dados e comunicação entre nós, podendo tornar difícil a migração de uma aplicação que não foi inicialmente desenhada para este tipo de cloud. É papel do arquiteto decidir quando determinada plataforma é apropriada para o projeto.

Essa desvantagem do acoplamento com a plataforma PaaS específica torna difícil também a portabilidade para outros provedores de cloud. Muitos serviços tentam amenizar o problema usando especificações consolidadas e portáveis, mas isto, às vezes, não é suficiente. No App Engine, o Datastore, seu banco de dados não relacional baseado no BigTable do Google, é bem diferente de se usar comparado a outros bancos de dados.

Tentando minimizar o acoplamento com a plataforma, é até oferecida uma implementação da JPA em cima do Datastore com certas limitações. Na prática, porém, as diferenças e limitações do Datastore acabam vazando para a aplicação, tornando quase impossível rodar uma aplicação normal já pronta com JPA sem fazer muitos ajustes. Embora válida, a preocupação com o acoplamento com uma plataforma específica pode ser bastante minimizada com um bom design orientado a objetos. Usando técnicas de separação de responsabilidades e bom encapsulamento, é possível isolar esse ponto negativo e usufruir de todas as outras vantagens do PaaS.

Não ter o controle total da infraestrutura também pode ser um problema, mesmo no IaaS. Detalhes da topologia da rede, configuração específica do hardware e até mesmo a geolocalização dos servidores podem trazer surpresas para uma aplicação. O Netfiix possui um dos mais conhecidos casos de sucesso na migração para o Amazon EC2,[116] porém, diversos problemas com a latência da rede, maior e mais variável que a que tinham anteriormente, forçou um acerto da granularidade dos serviços e do protocolo para reduzir o número de *roundtrips.*[98]

Há também quem chame serviços Web prontos como Gmail, Google Docs, Dropbox ou GitHub de cloud computing. Esses produtos são chamados de **SaaS** (*Software as a Service*), e são mais orientados ao usuário final. É a oferta de serviços dispensando instalações e configurações, facilitando para o usuário e permitindo acesso de qualquer lugar. Há também ofertas de serviços computacionais no modelo de SaaS, como o Pusher, uma API web para envio de mensagens disponibilizada como serviço para todo tipo de aplicações.

A aplicabilidade de cloud computing é bastante extensa, em geral com objetivos de escalabilidade, disponibilidade, corte de custos e facilidade de gerenciamento. Na Caelum, por exemplo, o Google App Engine é usado no site principal, não por motivos de escalabilidade, já que não é um site com muitos acessos. Mas há picos pontuais de acesso em que o volume de usuários aumenta mais de 10 vezes durante alguns mi-

nutos, quando são feitos grandes anúncios ou lançamentos. Cloud computing nos permite escalar elasticamente nesses momentos para atender aos novos usuários de maneira transparente e barata. Além disso, usar o App Engine nos trouxe mais disponibilidade e confiabilidade se comparado ao uso de um servidor próprio como antes. Diminuiu os custos com infraestrutura e simplificou bastante o processo de desenvolvimento e deploy.

Há ainda outros exemplos na Caelum. Um importante sistema da empresa possui um build complicado e longo, dada a complexidade do projeto e o número de testes. Durante muito tempo, convivemos com as dores de cabeça de gerenciar um servidor de integração contínua localmente.

Uma das questões mais problemáticas era o longo tempo de build, que engessava o processo de desenvolvimento. Com o EC2 da Amazon, rodamos os testes em paralelo em diversas instâncias do Hudson e Teamcity, agilizando bastante o processo de integração contínua. No campo de SaaS, usar o GitHub diminuiu drasticamente nossos problemas com infraestrutura e seus custos associados. Há diversos outros cenários e exemplos de todo tipo de empresas, sempre buscando facilidade para desenvolver, executar e escalar as aplicações, além de cortar custos e ter ambientes mais robustos, confiáveis e escaláveis.[115]

Existem críticas e questionamentos ao uso de cloud computing. A maior preocupação certamente é relacionada à segurança e à privacidade. Muitas empresas questionam se seus dados e sua aplicação estarão a salvo a partir do momento em que sua infraestrutura for terceirizada e depender de um provedor de cloud. Há duas questões envolvidas. A primeira é com relação à segurança contra ataques externos. Seria a infraestrutura do provedor mais segura que uma solução própria? Difícil saber, mas é fato que todos os bons provedores de cloud possuem equipes especializadas no gerenciamento de toda essas infraestrutura. Além disso, este é o coração dos negócios do provedor, e é certo supor que seu conhecimento nessa área tem grandes chances de ser maior que de empresas cuja infraestrutura própria seja apenas um suporte para sua atividade principal.

Outro ponto que ajuda a garantir a segurança contra ataques é a constante atualização dos softwares da infraestrutura usada. Quando temos máquinas próprias, é nosso dever atualizar quase que diariamente nosso sistema operacional, servidor Web, de banco de dados, etc. Na prática, é fácil ver empresas deixarem de lado o estafante trabalho de atualizar os softwares constantemente. Novamente, ao usar cloud, paramos de nos preocupar com isso e deixamos o trabalho na mão do provedor, que em geral possui políticas bastante rígidas e eficazes quanto a atualizações de segu-

rança, além de backups e redundância.

O segundo ponto de preocupação com a segurança é com relação ao provedor em si. Podemos estar mais seguros contra hackers ao usar cloud, mas quem garantirá a privacidade dos dados contra acessos do próprio provedor? Realmente não há solução, a não ser escolher provedores com boas políticas de privacidade e ações claras nessa direção, e confiar no provedor. Mas sempre há a possibilidade técnica de acesso local dos dados, como em qualquer outro serviço que usamos, tais como telefonia ou provedor de internet. É importante lembrar que todo negócio de um provedor de cloud é baseado na confiança de seus usuários, e eles certamente são os primeiros a perseguir a garantia de privacidade de seus clientes.

Um único episódio de violação de privacidade internamente no provedor seria capaz de quebrar a empresa. Mas, claro, há cenários em que os riscos envolvendo certos dados sigilosos são muito grandes, e o cloud computing clássico não é uma solução. Pensando nisso, há soluções de clouds privados que tentam trazer alguns dos benefícios do cloud computing para infraestruturas internas das empresas.

Outro ponto de preocupação é disponibilidade. Ao migrar para um provedor de cloud, fica-se completamente dependente de sua disponibilidade. E há casos emblemáticos de problemas nessa área, como a grande queda dos serviços da Amazon no data center da costa leste americana em Abril de 2011. Durante horas (e até dias, em alguns casos), diversos sites ficaram fora do ar, como Foursquare e Reddit. Já outros, como o Netflix, que haviam se preocupado com redundância entre data centers, conseguiram se manter no ar. Mas esse episódio, junto com diversos outros menores em praticamente todos os provedores, levanta a questão sobre a dependência da disponibilidade do provedor.

O fato é que 100% de disponibilidade é utópico, seja em infraestrutura própria ou para um provedor. A questão não é ter um sistema que não cai nunca, mas um que rapidamente se recupere de quedas. E a pergunta que se deve fazer é: qual a chance da minha infraestrutura própria cair em comparação com a chance de algum dos provedores ficar indisponível. De diversas formas diferentes, parece que o ambiente de alguns dos grandes e respeitados provedores de cloud tem maior chance de ficar no ar que uma infraestrutura própria, salvo talvez raras exceções de empresas com infraestruturas bem avançadas.

Considerações sobre escalabilidade

Jogar uma aplicação qualquer em cloud não funciona como solução mágica para escalabilidade. O cloud nos traz preparados para escalar elasticamente todo o hard-

ware, e, no caso de PaaS, a plataforma de software. Não há tanta preocupação com máquinas, softwares complicados para escalar, e até mesmo a configuração de novos nós de um cluster fica bastante facilitada. Mas escalabilidade é uma questão arquitetural, e as soluções para atacá-la costumam adicionar certa complexidade ao projeto. Não se deve adotar uma solução arrojada de escalabilidade, a menos que ela se torne necessária na prática.[106]

A dificuldade de se escalar uma aplicação envolve principalmente seu estado, seus dados. Como garantir que o sistema aguenta o volume de leituras necessário? Como paralelizar escrita de dados? Como replicar os pontos de acesso a dados sem perder a consistência? Como garantir a disponibilidade?

Quando se fala de dados, não é apenas sobre bancos e a persistência. Há também o estado dos servidores, como seus objetos internos ou as sessões do usuário. Ao criar um cluster, é importante decidir como mantê-las. Usar replicação total garante que todas as máquinas estão preparadas para tratar requisições de qualquer usuário, e que as informações não serão perdidas quando uma máquina cair. Mas, ao mesmo tempo, implicam uma carga muito maior para os servidores, o que pode ser inviável em clusters muitos grandes, mesmo que a atualização entre eles seja feita de maneira assíncrona, onerando a consistência. Por outro lado, existem também as *sticky sessions*, nas quais o mesmo usuário é sempre direcionado a um mesmo nó. Isso tira a carga da replicação, mas dificulta um balanceamento uniforme das requisições, além de não evitar perda de dados no caso de indisponibilidade. Muitas vezes, a solução acaba sendo um híbrido dessas duas abordagens, ou jogar as sessões para o banco de dados se este já estiver com sua escalabilidade resolvida. Ou, ainda, até chegar à solução mais extrema e ter uma aplicação *stateless* em relação ao usuário, altamente escalável, mas nem sempre viável.

Não é trivial escolher como montar um cluster de máquinas para o banco de dados apropriadamente para obter leituras e escritas de forma a ganhar escalabilidade e disponibilidade, ponderando a consistência. Em bancos relacionais é comum utilizar uma topologia *master-slave*, na qual uma máquina principal replica todo seu conteúdo para um ou mais *slaves*. Enquanto toda escrita é feita no *master*, por manter dados idênticos em mais de um lugar ao mesmo tempo, a leitura pode ser distribuída entre todas as outras máquinas. Isso resolve o problema de muitas leituras, mas não é solução se tivermos um grande volume de escritas, além de possuir um único ponto de falha que afeta a disponibilidade. Há algoritmos que suportam escritas em diferentes nós, mas há sérias complicações técnicas, principalmente ligadas à consistência. E, uma vez que é difícil e lento implementar transações distribuídas,

evita-se sua utilização em sistemas de grande escalabilidade[174] (Figura 6.18).
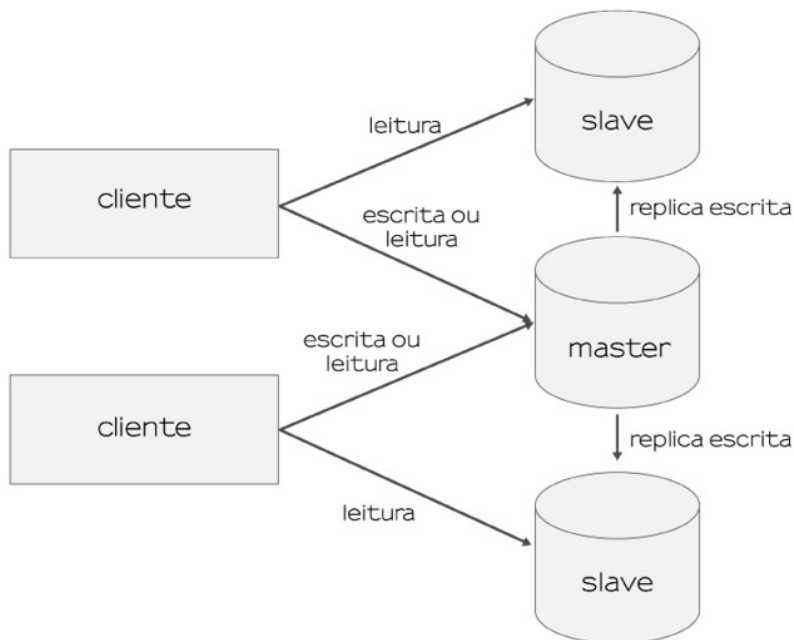
Figura 6.18: Topologia master-slave em bancos de dados.

À medida que os dados atingem volumes gigantescos, torna-se inviável mantê-los todos em uma única base, como uma única tabela em um banco relacional. Atualizações ficam cada vez mais lentas e os índices cada vez mais ineficazes. Torna-se também muito caro replicá-lo por completo. A solução passa a ser particionar os dados em mais tabelas e bancos de dados. Pode-se quebrar as linhas em tabelas diferentes, algo conhecido como particionamento horizontal, ou *sharding*. Uma dificuldade é como decidir os pontos de divisão. Devemos dividir os clientes de A-M em uma tabela, e de N-Z em outra? Pode não ser uma divisão balanceada; para isto, há algoritmos para particionar de maneira mais distribuída com hashes ou ids. O Twitter, por exemplo, utiliza outra chave comum de particionamento, a data de inserção de uma informação; dados antigos são retornados com frequência baixa e podem ser armazenados em listagens distintas.[106] A chave vai depender da aplicação. Outra abordagem para particionamento seria a vertical, com tabelas com menos colunas, que passam a estar mais espalhadas para dividir os dados (Figura 6.19).
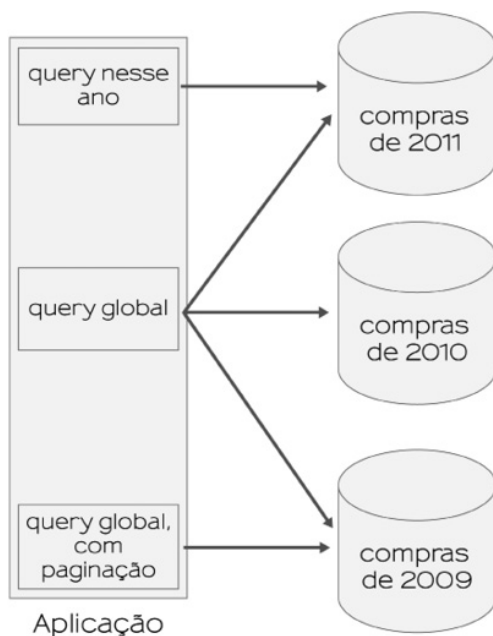
Figura 6.19: Particionamento horizontal de dados pela data.

Abrir mão de conceitos tradicionais relacionados a bancos de dados é outra solução. Em situações de alta escalabilidade, desnormalizar os dados passa a ser uma opção. Isso significa duplicar dados em diferentes lugares para facilitar leituras e sobrecarregar menos o banco de dados, mesmo dificultando a atualização dos dados para a aplicação. Ou, ainda, guardar resultados de agregações e agrupamentos, em vez de usar o banco para calculá-los dinamicamente. Em todos os casos, há um trade-off claro entre escalabilidade e uma maior complexidade do modelo de dados e do código da aplicação para manipulá-los.

A solução mais usada e recomendada para escalabilidade é o uso de caches para suportar mais leituras e evitar o acesso ao banco de dados. Usa-se o cache de segundo nível do Hibernate, como vimos no tópico 6.3, e caches distribuídos, como o Memcache, com esta finalidade. Embora não resolvam o problema de escalar escrita, os caches são essenciais para escalabilidade, tendo em vista que na maioria das aplicações há mais leituras que escritas.

Mas usar caches ou replicação sempre levanta o ponto da consistência dos dados, como, por exemplo, garantir que não se leia dados antigos. Porém, consistência é um termo difícil de se definir. Pode ser bastante aceitável que, depois de uma escrita, os

dados só estejam visíveis para leituras de todos após alguns segundos. Em outras aplicações, isto é inadmissível. Um portal de notícias talvez não exija que as notícias na home page apareçam no exato instante de sua publicação, mas um sistema de cotações da bolsa de valores pode não ter esta flexibilidade. Há soluções de bancos de dados que flexibilizam a garantia de consistência tradicional em bancos relacionais. Com isso, ganha-se bastante possibilidade para escalar.

Três características dos bancos e das aplicações são importantes para escolher a forma de abordar a escalabilidade: consistência, disponibilidade ou tolerância a falhas e particionamentos. O Teorema CAP[26],[197] postula que só é possível ter duas dessas garantias simultaneamente. Cabe ao arquiteto avaliar quais delas trarão mais valor à aplicação. Tal consideração é ainda bastante simplista, afinal, o que é consistência para sua aplicação? Se isso significa não ver dados desatualizados depois de 10 segundos de uma modificação, pode ser possível atingir os três requisitos. O mesmo questionamento ocorre para as outras duas características.

Muito se questiona hoje em dia sobre qual banco de dados usar. Anteriormente, bancos relacionais eram praticamente obrigatórios, mas o crescimento da Web e da necessidade de soluções mais robustas de escalabilidade fizeram com que novas opções surgissem. O movimento NoSQL procura mostrar outras abordagens a bancos de dados com soluções não relacionais, como key-value stores, bancos baseados em grafos, orientados a documentos, entre outros. Soluções de cloud computing são baseadas em bancos não relacionais, como o Bigtable, do Google App Engine, e o SimpleDB, da Amazon. Há também muitos disponíveis para uso, como Apache HBase, MongoDB, Apache Cassandra, criado pelo Facebook, ou Voldemort, criado pelo LinkedIn.

No Google App Engine, por exemplo, há duas soluções de persistência baseadas no BigTable.[213] Seu datastore padrão é baseado em replicação assíncrona master/slave e garante consistência, mas não disponibilidade. Outra opção é o *High Replication Datastore*, que garante disponibilidade, mas enfraquece as garantias de consistência, além de ser mais custoso de usar. O desenvolvedor precisa optar pela solução que for mais adequada para a aplicação. Uma decisão frequente na hora de escolher um banco NoSQL é abrir mão da consistência e ter um sistema *eventualmente consistente*. Abre-se mão das tradicionais garantias do ACID para adquirir o BASE (*Basis Available, Soft state, Eventual consistent*).[162]

Capítulo 7

Integração de sistemas na Web e REST

Sistemas sem uma *API* externa vivem isolados em silos de dados, algo cada vez mais raro, já que a necessidade de se integrar com outros sistemas é crescente. Por isso, ao desenvolver uma nova aplicação, pensa-se em como será sua integração com outras no futuro.

Existem diversas opções de integração para sistemas distribuídos, cada uma apresentando níveis de acoplamento e coesão distintos, centralizando ou distribuindo o controle sobre os processos que ocorrem, abordagens focadas em sistemas homogêneos ou nas quais os clientes serão aplicações desconhecidas de terceiros. A maneira escolhida para integrar os sistemas hoje implicará os custos de manutenção do software durante os próximos anos, portanto, tal escolha deve ser feita analisando-se os *trade-offs* de cada solução.

Entre as várias possibilidades, têm ganhado destaque as que facilitam atingir requisitos não funcionais sem um trabalho extra, além de possibilitar uma evolução

dos serviços evitando a quebra de clientes. É nesta linha de pensamento que a Web (com HTTP e hipermídia) como plataforma e a abordagem REST se sobressaem. Infraestrutura já existente, como caches HTTP, proxies, balanceadores e ferramentas de monitorações passam a funcionar como um middleware mais transparente e simples.

7.1 Princípios de integração de sistemas na Web

Sistemas que não apresentam pontos de integração estão fadados a um esforço muito grande de manutenção[175], uma vez que os mercados corporativo e Web caminham para a divisão de responsabilidades entre pequenos sistemas, na contramão da antiga abordagem centralizadora. Dois sistemas quaisquer integrados podem ser vistos como parte de um único maior, cada um escrito com linguagens e pilhas de tecnologias possivelmente diferentes[12]. O meio de comunicação entre eles é um aspecto fundamental na arquitetura.

No passado, a pluralidade nos padrões e protocolos implicava a necessidade de implementar diversos requisitos, como segurança, autenticação, autorização, cache, etc., que deviam ser suportados em grande parte deles. Tanenbaum ironiza que a vantagem dos padrões consiste na existência de diversos padrões para escolhermos[195], enquanto John Sowa diz, em sua *Law of Standards* (Lei dos Padrões), que "*toda vez que uma organização desenvolve um novo sistema como um padrão para X, o resultado principal é a ampla adoção de um sistema mais simples como o real padrão para X*"[188]. Com o tempo, surgiu a necessidade de uso de um protocolo para transferência de dados que já suportasse naturalmente esses requisitos mencionados.

Hoje, o protocolo padrão para troca de informações é o HTTP. A porta 80 é bem-vinda pelos firewalls, e, ao mesmo tempo, existem diversas ferramentas, clientes, servidores e intermediários que suportam o protocolo. Muitas dessas ferramentas trazem soluções, por exemplo, para problemas de performance e escalabilidade no uso do HTTP. Além disso, é possível desenvolver sistemas Web nas mais diversas linguagens de programação.

Utilizar o HTTP como base para a comunicação foi o caminho encontrado pelas organizações para uma nova geração de padrões de comunicação. A evolução do uso de hipermídia e da HTTP permitiu que este superasse muitos outros protocolos inventados para indústrias específicas[168].

Representando os dados e as operações

Desde que a ideia de integrar sistemas na Web surgiu, para facilitar o processo de desenvolvimento e consumo dos serviços, o foco voltou-se para a utilização de XML puro, conhecido como *POX* (*plain old XML*), para a execução de chamadas remotas.

Em muitos casos, esses XMLs são trocados entre consumidor (cliente, ou *consumer*) e serviço através de requisições do tipo *POST*, utilizando uma única URI para acesso. No exemplo a seguir, um sistema utiliza a URI `/servicos` para buscar as informações de um cliente, adicionando no corpo da requisição qual a operação exata que o cliente deseja executar:

```
POST /servicos HTTP/1.1
Host: arquiteturajava.com.br
Content-Type: application/xml
Content-Length: 91

<acao>
<codigo>busca_cliente</codigo>
<parametros>
<id>15</id>
</parametros>
</acao>
```

O exemplo a seguir mostra um XML de retorno, descrevendo um cliente:

```
<cliente>
<nome>Guilherme Silveira</nome>
<empresa>Loja Caelum</empresa>
</cliente>
```

Para ações com resultados, como a criação de um cliente, o servidor costuma devolver o resultado da operação dentro do corpo da resposta, não utilizando o *response code* do protocolo HTTP, mas uma mensagem própria:

```
HTTP/1.1 200 OK
Date: Sun, 18 Sep 2011 18:09:00 GMT
Content-type: application/XML

<resultado>OK</resultado>
```

Um protocolo foi até mesmo criado, chamado *XML-RPC*, padronizando esta comunicação. Na prática, a maior parte dos sistemas on-line que utilizam XML ou JSON não adota essa especificação, utilizando seu próprio formato e padrões.

Essa abordagem envolve o mínimo de utilização do protocolo HTTP, somente para *tunneling*[168], mas ganha muito na facilidade de implementação. Em algumas variações, utiliza-se *query parameters* para decidir qual ação será executada ou até mesmo para passar também os parâmetros de uma requisição.

Um serviço assim pode ser implementado com qualquer framework Web e uma API de serialização de XML, como *XStream*, no exemplo a seguir, ou binding, como o *JAX-B*.

```
@XStreamAlias("cliente")
public class Cliente {
    private String nome, empresa;

    public Cliente(String nome, String empresa) {
        this.nome = nome;
        this.empresa = empresa;
    }

    public String getNome() {
        return nome;
    }
    public String getEmpresa() {
        return empresa;
    }
}

XStream xstream = new XStream();
xstream.processAnnotations(Cliente.class);

Cliente cliente = new Cliente("Guilherme Silveira", "Loja Caelum");
System.out.println(xstream.toXML(cliente));
```

O resultado do exemplo acima é o XML:

```
<cliente>
<nome>Guilherme Silveira</nome>
<empresa>Loja Caelum</empresa>
</cliente>
```

Os clientes acessam os servidores através de *APIs* HTTP, como *Apache Http Client*, e novamente uma ferramenta de serialização ou binding:

```
String uri = "http://arquiteturajava.com.br/loja/cliente/7465";

HttpClient client = new DefaultHttpClient();
HttpGet get = new HttpGet(uri);
HttpResponse response = client.execute(get);

System.out.println(response.getStatusLine().getStatusCode());

HttpEntity entity = response.getEntity();
InputStream input = entity.getContent();

Cliente cliente = (Cliente) xstream.fromXML(input);
System.out.println(cliente.getNome());
```

HTTP

Adotando a característica do protocolo HTTP de que toda *URI* identifica algo, alguns sistemas utilizam diversas *URIs* e um único método, em geral *POST*. Em um próximo passo de adoção do protocolo, diversos métodos são utilizados para facilitar a identificação da tarefa sendo executada, como *GET*, para obter informações, *POST*, para criar, *PUT*, para alterar, *DELETE*, para remover, etc. Isso permite que o sistema como um todo se beneficie das vantagens de cada método onde o mesmo for adequado, como ao cachear resultados de uma query executada através de uma requisição *GET*.

Com a popularização de outros formatos que fazem ou não uso de schemas, como o *JSON*, diversos serviços expostos na Web passaram a visar à simplicidade, expressividade, ou, ainda, aumentar a compatibilidade entre versões diferentes dos serviços.

A mescla de diversos sistemas Web para a criação de um novo (*mashups*) foi facilitada por uma nova geração de ferramentas e serviços Web. Grande parte delas foi criada em cima de *JSON*, distribuindo o conteúdo dos sistema originais. É comum encontrar fotos do Flickr, vídeos do YouTube, mapas do Google e grupos do Facebook nas mais diversas aplicações on-line, tudo isso resultado dessa integração simples na Web, vindas das ideias de *XML-RPC*.

Frameworks, como o *VRaptor*, apresentam convenções que facilitam a criação de serviços que geram resultados em diversos formatos, dando acesso às informações do protocolo HTTP que este desejar. Outras soluções, como a especificação JAX-RS, que veremos adiante, suportam, sem convenções, a criação de tais serviços.

7.2 Padronizações, contratos rígidos e SOAP

Baseado em RPC, o SOAP é um envelope que ganhou aceitação por poder utilizar a pilha de tecnologias que já se encontrava onipresente nas plataformas de programação, como a transferência de conteúdo XML via HTTP. Visando à independência do protocolo utilizado, as mensagens poderiam ser transferidas também através de SMTP, entre outros, dando uma grande liberdade de escolha para o protocolo de "baixo nível". Ao abstrair o protocolo, é necessário garantir um mínimo de padronização sobre a operação a ser executada, como o formato da mensagem, questões de segurança e cache.

Apesar da generalização da abordagem, na prática, a maior parte dos sistemas utiliza o *tunneling* via HTTP (*Web Services*). Mas os protocolos são heterogêneos, cada um deles possui restrições e funcionalidades distintas, como, por exemplo, o suporte nativo a assincronismo no JMS, ou o suporte a autenticação, cache e compressão no HTTP.

Para garantir que características não comuns a todos os protocolos sejam suportadas de maneira homogênea, foi necessária a criação de extensões que padronizaram cada uma dessas características, normalmente configuradas através dos cabeçalhos de mensagens existentes no SOAP[209]. Por exemplo, no caso de autenticação, criou-se uma extensão chamada *WS-Security*, permitindo que a utilização de SOAP funcionasse independentemente do uso de determinado protocolo para *tunneling*. Hoje, existem diversas extensões, como *WS-Addressing*, *WS-Notification* e outros[104], frequentemente referenciadas conjuntamente pela sigla *WS-** (Figura 7.1).
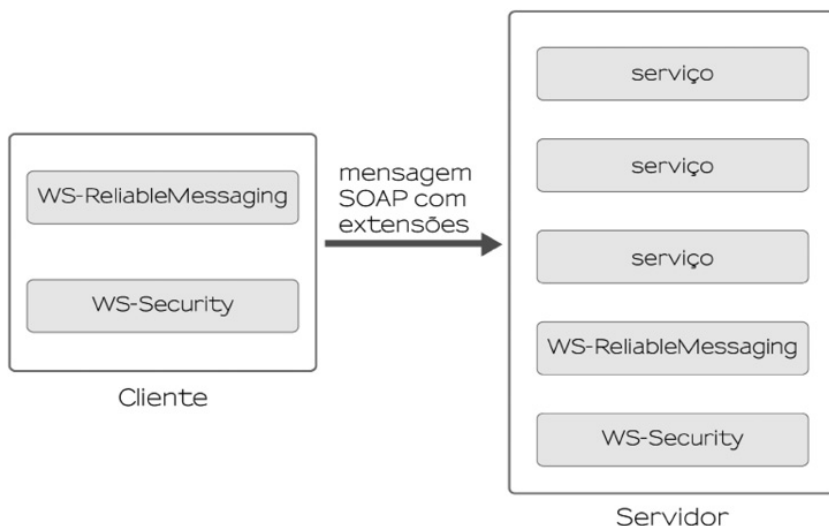
Figura 7.1: Exemplo possível de stack para troca de mensagem com SOAP.

SOAP na prática

Para que um serviço seja consumido, o cliente precisa conhecer as diversas estruturas utilizadas e disponibilizadas pelo sistema, como os *schemas* adotados, serviços disponíveis e seus pontos de acesso. Essas definições são feitas através da interface remota do serviço, descrito, por sua vez, através da especificação de um *WSDL*.

Apesar de a versão 1.2, de 2003, ser a primeira recomendada pelo *W3C*, a maior parte das ferramentas suporta somente a versão 1.1, como no caso do WS-BPEL 2.0[11]. O exemplo a seguir mostra parte de um arquivo compatível com *WSDL* 1.1, uma das maneiras de definir os possíveis *schemas* ou tipos:

```
<wsdl:types>
    <s:schema elementFormDefault="qualified"
        targetNamespace="http://arquiteturajava.com.br/bolsa">
        <s:element name="PegarValorAtual">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="0" maxOccurs="1"
                        name="simbolo" type="s:string" />
                </s:sequence>
            </s:complexType>
        </s:element>
```

```
        <s:element name="PegarValorAtualResponse">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="1" maxOccurs="1"
                        name="PegarValorAtualResult" type="s:float" />
                </s:sequence>
            </s:complexType>
        </s:element>
    </s:schema>
</wsdl:types>
```

A definição de uma operação é feita separadamente, junto com a descrição dos tipos esperados como entrada e saída. No exemplo a seguir, a operação `PegaValorAtual` recebe uma estrutura `PegaValorAtualSoapIn` e devolve `PegaValorAtualSoapOut`:

```
<wsdl:message name="PegarValorAtualSoapIn">
    <wsdl:part name="parameters" element="tns:PegarValorAtual" />
</wsdl:message>
<wsdl:message name="PegarValorAtualSoapOut">
    <wsdl:part name="parameters" element="tns:PegarValorAtualResponse" />
</wsdl:message>
<wsdl:portType name="BolsaDeValoresServiceSoap">
    <wsdl:operation name="PegarValorAtual">
        <wsdl:input message="tns:PegarValorAtualSoapIn" />
        <wsdl:output message="tns:PegarValorAtualSoapOut" />
    </wsdl:operation>
</wsdl:portType>
```

Apesar de não ser mandatório, o *WSDL* é normalmente utilizado com SOAP, e é a tag binding da *WSDL Binding Extension* que define esse acoplamento, além de indicar qual o protocolo utilizado para *tunneling*:

```
<soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
```

Por fim, são definidas as URIs a serem acessadas:

```
<wsdl:binding name="BolsaDeValoresServiceSoap12"
    type="tns:BolsaDeValoresServiceSoap">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="PegarValorAtual">
        <soap12:operation style="document"
```

```
            soapAction="http://arquiteturajava.com.br/bolsa/PegarValorAtual" />
        <wsdl:input>
            <soap12:body use="literal" />
        </wsdl:input>
        <wsdl:output>
            <soap12:body use="literal" />
        </wsdl:output>
    </wsdl:operation>
</wsdl:binding>
<wsdl:service name="BolsaDeValoresService">
    <wsdl:port name="BolsaDeValoresServiceSoap12"
        binding="tns:BolsaDeValoresServiceSoap12">
        <soap12:address
            location="http://arquiteturajava.com.br/bolsa/BolsaDeValoresService"/>
    </wsdl:port>
</wsdl:service>
```

O desenvolvedor pode utilizar ferramentas, como o `wsimport` do *JAX-WS*, para gerar *stubs* que abstraem e simplificam o acesso aos serviços expostos através de SOAP. Esses *stubs* costumam ser classes que delegam invocações para bibliotecas HTTP acessarem o serviço. O resultado é a criação de dezenas de classes, que mapeiam essas definições para estruturas de dados da linguagem escolhida, como value objects e entidades. Mudanças no WSDL implicam a necessidade de reanálise e provável nova geração de *stubs*.

```
@WebService
public class PagamentoService {

    @WebMethod
    @WebResult(name="pagamentos")
    public List<Pagamento> getPagamentosPorCliente(
        @WebParam(name="cliente") Cliente cliente) {
        List<Pagamento> pagamentos = //lógica
        return pagamentos;
    }

}
```

Sobre a adoção do SOAP

A existência de tantas extensões reforça a ideia de que SOAP deixou de lado pon-

tos que o protocolo HTTP já suportara. Cada abstração introduzida através de uma extensão aumenta a complexidade do sistema, portanto, desenvolvedores e arquitetos devem pesar o benefício dessas decisões contra a desvantagem que elas apresentam.

Segundo James Snell, que foi responsável por parte dessas tecnologias quando na Microsoft, "*SOAP começou com um caminho simples e direto para fazer RPC com tunneling via HTTP, com um mero POST, mas cresceu para suportar nomes de métodos em headers*"[187]. Segundo Stefan Tilkov, o WS-* começou com um pequena pilha de padronizações e cresceu para um alto número de complexas especificações, que dificultam a utilização da tecnologia[202]. Historicamente, uma evolução sobre os modelos de integração de sistemas anteriores, o SOAP, após mais de 10 anos de existência, apresenta hoje altos custos e tem se mostrado de difícil manutenção devido à alta complexidade e à abstração do protocolo utilizado.

O mercado tem visto com cada vez mais cautela a adoção de SOAP como solução técnica, especialmente comparando-o à possibilidade de utilizar a Web como plataforma de integração. O Google, por exemplo, abandonou a busca através de SOAP em 2009[198]. Stefan Tilkov diz que "*no momento atual, as vantagens técnicas deixaram de valer a pena, somente recomendamos o uso de Web services tradicionais via SOAP quando questões políticas estão envolvidas, como, por exemplo, acesso a um sistema SAP que já possui o serviço disponibilizado*"[200]. James Snell, enquanto trabalhava na Microsoft, desenvolvendo WS-*, diss: "nunca trabalhei em uma aplicação que resolvesse uma necessidade real de negócio e, agora, trabalhando junto a dezenas de milhares de desenvolvedores da IBM, nunca passei por uma situação onde WS-* fosse uma solução que se encaixasse"[105].

Na tentativa de resolver parte dos problemas, a versão 1.2 do SOAP[210] se aproxima mais do protocolo HTTP, deixando de utilizá-lo somente para *tunneling*, ou execução de chamadas RPC. Tais mudanças, como o suporte a métodos que não somente o POST, facilitaram a utilização de layers intermediários, como, por exemplo, cache, através dos headers HTTP.

Muitas ferramentas não suportam ou não encorajam os desenvolvedores a utilizar os detalhes das versões mais novas do SOAP, consequentemente perdendo as vantagens da utilização do HTTP. O mercado que utiliza WS-* ainda não adotou tais mudanças, limitando o uso do protocolo. O mapeamento ao protocolo ainda é muito superficial, e a imagem de que o mesmo deve ser usado para RPC se mantém: "*Um dos objetivos do design do SOAP é encapsular e fazer chamadas RPC usando a extensibilidade e flexibilidade do XML*" [210].

7.3 Evite quebrar compatibilidade em seus serviços

Parte do segredo do baixo custo de manutenção está em manter a coesão e diminuir o acoplamento entre partes de um sistema. Ao aplicar este conceito na integração entre dois sistemas, os mesmos devem minimizar os pontos de acoplamento para maximizar a capacidade de mudança sem quebra de compatibilidade. O processo de criar uma nova versão de um serviço é chamado versionamento (*versioning*), e pode ou não manter compatibilidade com as anteriores. Uma implementação que obriga a atualização de clientes toda vez que uma nova versão do serviço é lançada apresenta alto nível de acoplamento e alto custo de manutenção.

Quanto mais genéricos os clientes, menor o acoplamento com um serviço específico e, portanto, menor a necessidade de manutenção dos mesmos, dada uma atualização do servidor acessado. Diversas técnicas podem ser aplicadas em qualquer tipo de integração de sistemas e visam minimizar a necessidade dessas atualizações.

Validação de estrutura e esquemas

Ao receber os dados de um outro sistema, é possível validar sua estrutura por completo, com um controle mais rígido sobre o que a outra ponta pode fazer. Nessas situações extremas, validam-se até mesmo os dados que não serão utilizados, mas descartados. Um exemplo de tal abordagem surge ao validar um XML contra um *schema* completo, mesmo utilizando somente uma parte do conteúdo recebido.

O exemplo a seguir mostra um schema que define uma estrutura para representar ordens de compra, incluindo o destino e uma lista de produtos:

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="compra">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="cliente" use="required">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="id" type="xs:string"/>
              <xs:element name="nome" type="xs:string"/>
              <xs:element name="dataDeNascimento" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
```

```
        <xs:element name="produto" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="id" type="xs:string"/>
              <xs:element name="quantidade" type="xs:positiveInteger"/>
              <xs:element name="preco" type="xs:decimal"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Além da validação completa contra o esquema, uma ferramenta de binding que não ignora campos desconhecidos implica o mesmo tipo de acoplamento; qualquer mudança no esquema leva à necessidade de atualização de toda a base de clientes.

Nesses casos, um campo novo adicionado em uma representação XML quebra a compatibilidade, pois quem os consome não esperava esse novo valor. O problema surge quando os clientes são acoplados a um contrato fixo, uma estrutura completa e imutável. *Schemas* são maneiras de garantir uma estrutura, mas seus clientes devem minimizar o acoplamento, permitindo ao máximo sua evolução.

O exemplo a seguir mostra uma possível evolução do *schema*, adicionando um campo novo, contendo o CPF do cliente e removendo a data de nascimento:

```
<xs:element name="cliente" use="required">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id" type="xs:string" />
      <xs:element name="nome" type="xs:string" />
      <xs:element name="cpf" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Uma aplicação tradicional validará o campo dataDeNascimento independentemente se este for ou não utilizado. As classes geradas também possuirão um campo para a `dataDeNascimento`. Por isso, o resultado da remoção desse campo é um erro de validação, e a quebra da aplicação. Uma vez que ela não utilizava os dados, mas deixou de funcionar, esse acoplamento a uma definição rígida é desnecessário.

É importante validar somente a parte do XML que será utilizada, ignorando quaisquer mudanças de estrutura naquilo que não é necessário. É preciso seguir à risca o *schema* no momento de gerar os dados a serem enviados. Seguindo essas duas regras, as aplicações cliente possuem as garantias necessárias, mantendo um acoplamento mais flexível ao contrato.

O *WSDL*, por exemplo, não obriga, mas incentiva a validação completa através de *schemas*, reforçando esse acoplamento.

Para executar validações parciais é possível utilizar ferramentas como o *RelaxNG*, pela qual o servidor será capaz de adicionar ou remover dados sem quebrar todos os clientes existentes. Em vez de uma consequência tão negativa, somente aqueles clientes que utilizavam campos removidos precisarão ser atualizados. Esta é a base do padrão *Must Ignore*, criado por David Orchard em 2003[150], que descreve práticas para facilitar a evolução de um *schema XML*, assunto também mencionado por Ian Robinson, com práticas voltadas a *backward* e *forward compatibility*[55]. Essas práticas podem ser utilizadas com qualquer tipo de serviço que consuma mensagens validáveis através de um *schema*.

Há ainda formatos mais amigáveis à evolução do schema, como o *protocol buffers* (`application/x-protobuf`), criado pelo Google[84]. Para não quebrar os clientes, eles são capazes de fornecer valores padrão para campos que foram removidos. O exemplo a seguir mostra a definição de um esquema de mensagem de cliente com um campo nome e pais, onde o país padrão é Brasil. Caso em uma nova versão do sistema um campo não seja enviado, basta alterar o esquema para definir um valor padrão, podendo assim manter a compatibilidade com versões anteriores.

```
message Cliente {
    required string nome = 1;
    optional string pais = 2 [default = ‘‘brasil"];
}
```

Mantendo diversas versões incompatíveis no ar

Em alguns momentos, é necessário quebrar a compatibilidade com a versão anterior, e fica difícil minimizar o custo de manutenção daqueles que já consomem os serviços. É possível criar novas *URIs* de acesso para cada release de versão incompatível do serviço. Desta maneira, durante, por exemplo, dois anos, o servidor se compromete a manter compatibilidade no conjunto de serviços expostos, e, após esse tempo, estes já estarão substituídos e serão removidos, permitindo que os clientes façam suas alterações em ciclos maiores. Esta abordagem, apesar de aliviar o

custo na manutenção nos clientes, não remove a necessidade de sua adaptação a cada ciclo de versionamento.

7.4 Princípios do SOA

Após anos de discussão sobre o que seria exatamente uma arquitetura orientada a serviços (*SOA*), e vindo de uma década quando a adoção de SOAP era considerada a única solução para essa arquitetura, em 2009 foi criado o *SOA Manifesto*[56], um conjunto de características importantes para esse tipo de arquitetura baseado na visão do mercado naquele momento. Segundo o SOA Manifesto, é possível implementar uma arquitetura em cima de CORBA, SOAP, DCOM, REST, etc.

Nessa linha do *SOA*, a granularidade dos serviços deve permitir o reúso por diversos clientes, além da criação de novos serviços através da composição de outros. Por exemplo, para completar um processo (*workflow*) de compra, serviços são pensados como funções que atingem pequenos objetivos dentro do fluxo, como `PesquisarProdutos`, `CriarCompra`, `AlterarCompra`, `CancelarCompra`, `AlterarDestino`, `VerificarCompra`. Os Web Services da Amazon, por exemplo, apresentam esse tipo de granularidade, como no caso da *Product Advertising API*, que fornece acesso à busca de produtos no catálogo interno. Por possuírem uma granularidade menor, é possível reutilizar esses serviços em diversos tipos de aplicativos, desde um sistema Desktop que mostra o rastreamento de suas compras, passando por plugins para calendários, que mostram a data esperada de chegada da entrega, até clientes que automatizam compras, como livros para colecionadores que estão com o preço abaixo do mercado (Figura 7.2).
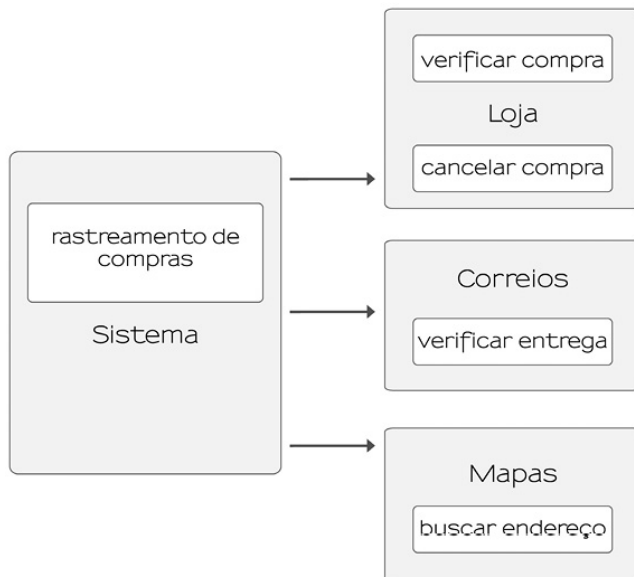
Figura 7.2: Um serviço de rastreamento de compras composto da utilização de diversos outros serviços, na mesma ou em outras máquinas.

É importante que seja possível descobrir detalhes sobre o serviço, além de utilizar contratos formais que os descrevam, como, por exemplo, SOAP através de *WSDL* e *schemas xml*. Os serviços abstraem de seus clientes os detalhes internos de suas implementações, funcionando como a interface de comunicação, encapsulando sistemas e processos possivelmente preexistentes. Por fim, serviços implementados com as ideias de SOA apresentam mais benefícios para a aplicação quando não mantêm nenhuma informação do cliente atual no servidor, apesar de não ser um requerimento para o desenvolvimento do sistema.

Componha serviços

Uma vez que centenas de serviços estarão expostos, é necessário controlar como será feita a interação entre eles. A criação de novos sistemas e serviços baseados naqueles preexistentes pode ser feita através de um código que acessa e controla todo o fluxo (orquestra, *orchestration*) de um grande processo (um *workflow*).

A Figura 7.3 exemplifica um pedido de compra, no qual há um processo de orçamento com diversos fornecedores externos, a busca do mais barato entre eles e o

pedido para liberação de verba.
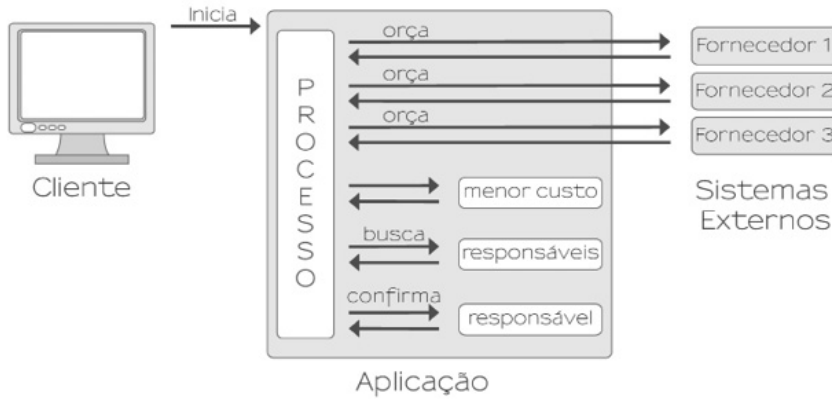
Figura 7.3: Exemplo de processo de pedido de compra.

O exemplo a seguir mostra como é possível descrever um processo de pedido de compra por um usuário cujo orçamento será feito com diversos fornecedores:

```
Pedido pedido = comprador.pede(produtos);

List<Orcamento> orcamentos = new ArrayList<Orcamento>();
for (Fornecedor fornecedor : fornecedores) {
    orcamentos.add(fornecedor.getOrcamento(pedido));
}
```

O menor orçamento é escolhido e o pedido de aprovação, enviado para o responsável. Quando esse pedido é respondido, um *callback* é feito assincronamente para o `OrcamentoAnalisado`:

```
Comparator<Orcamento> comparador = new MaisBaratoComparator();
Orcamento orcamento = Collections.min(orcamentos, comparador);

double valor = orcamento.getValor();
Usuario responsavel = comprador.responsavelPorAprovacaoPara(valor);
responsavel.pedeAprovacao(orcamento, new OrcamentoAnalisado());
```

Este primeiro passo, que descreve o fluxo de compra, pode ser escrito de maneira genérica:

```
public class PedidoDeCompra implements Passo {

    private final List<Fornecedor> fornecedores;

    public PedidoDeCompra(List<Fornecedor> fornecedores) {
        this.fornecedores = fornecedores;
    }

    public void executa(Map<String, Object> args) {

        Usuario comprador = (Usuario) args.get(‘‘comprador");
        Produtos produtos = (Produtos) args.get(‘‘produtos");
        Pedido pedido = comprador.pede(produtos);

        List<Orcamento> orcamentos = new ArrayList<Orcamento>();
        for (Fornecedor fornecedor : fornecedores) {
            orcamentos.add(fornecedor.getOrcamento(pedido));
        }
        Comparator<Orcamento> comparador = new MaisBaratoComparator();
        Orcamento orcamento = Collections.min(orcamentos, comparador);
        double valor = orcamento.getValor();

        Usuario usuario = comprador.responsavelPorAprovacaoPara(valor);
        usuario.pedeAprovacao(orcamento, new OrcamentoAnalisado());
    }
}
```

Após o responsável tomar uma ação, aprovando ou recusando o orçamento, o processo continua, finalmente efetuando o pedido de venda ou notificando seu cancelamento.

```
public class OrcamentoAnalisado implements Passo {

    private final Emails emails;

    public OrcamentoAnalisado(Emails emails) {
        this.emails = emails;
    }

    public void executa(Map<String, Object> args) {
        Confirmacao resposta = (Confirmacao) args.get(‘‘resposta");
```

```
        if (resposta.isPositiva()) {
            resposta.getOrcamento().executa();
        } else {
            emails.notificaRecusa(resposta.getPedido());
        }
    }
}
```

Diversas características do fluxo descrito anteriormente são padrões comuns (como os *Enterprise Integration Patterns*)[99] que aparecem ao descrever processos de integração de sistemas. Ferramentas, como o Apache Camel, implementam APIs baseadas em *method chaining* ou XML para trabalhar com esses processos. Por exemplo, o conteúdo do laço for poderia ser executado em paralelo (padrão chamado de *Splitter*), e as respostas devem ser aguardadas para então serem combinadas (*Agregator*).

Com algumas técnicas de design, o exemplo anterior pode ser encapsulado em uma *DSL*, reforçando o vocabulário do domínio utilizado:

```
public class PedidoDeCompra {

    private final List<Fornecedor> fornecedores;

    public PedidoDeCompra(List<Fornecedor> fornecedores) {
        this.fornecedores = fornecedores;
    }

    public void compra(Usuario comprador, List<Produto> produtos) {
        Pedido pedido = comprador.pede(produtos);
        Orcamento orcamento = fornecedores.quota(pedido, maisBarato());
        comprador.confirma(orcamento).em(OrcamentoAnalisado.class);
    }

}

public interface CallBack<T> {
    void processa(T argumento);
}

public class OrcamentoAnalisado implements CallBack<Confirmacao> {
```

```
    private final Emails emails;

    public OrcamentoAnalisado(Emails emails) {
        this.emails = emails;
    }
    public void processa(Confirmacao resposta) {
        resposta.executaOrcamento(emails);
    }

}
```

Organizar e coordenar a sequência de troca de informações entre serviços através de um ponto de código único é o papel de um orquestrador. Uma vez que todos os serviços estão descritos, a engine responsável pela troca de mensagens (*enterprise service bus*, ou *ESB*) coordena tudo o que será feito durante a execução de *workflows*.

Em caso de falha, é responsabilidade do sistema central mitigar o risco envolvido, por exemplo, tentando novamente após determinado período de tempo. Esse sistema pode ser encarregado de executar requisições a diversos serviços em paralelo e aguardar a resposta para então continuar no fluxo, entre outras tarefas. Fornecedores de ferramentas e soluções SOA apresentam produtos que orquestram serviços, visando facilitar a criação e manutenção de aplicações que centralizam o controle do processo.

Para um sistema que seja composto por mais de um serviço distinto, como a reserva de hotel e a de voo, cada um hospedado em empresas e servidores diferentes, é importante encontrar uma solução que coordene os casos de falha. Disparar as duas requisições em paralelo acelera o fluxo, mas o controle de falha na reserva do voo após a confirmação do hotel deve ser implementado de maneira a cancelar o hotel. Transações distribuídas como essas possuem os problemas de consistência e integridade naturais dessa arquitetura, e ferramentas como as de orquestração, junto com os diversos padrões WS-*, podem ajudar a lidar com eles, adicionando certa complexidade.

Serviços podem ser compostos programaticamente ou através de ferramentas. Não é a adoção de um produto de orquestração ou de geração de código que trará o sucesso a uma arquitetura orientada a serviços, mas, sim, sua compreensão técnica e sua adoção no momento adequado.

7.5 REST: arquitetura distribuída baseada em hipermídia

Na abordagem orientada a serviços tradicional, a tendência é forçar os desenvolvedores a criarem múltiplas operações, além de diversos tipos de dados. Por exemplo, um sistema simples, com tipos representando clientes, compras, produtos e pagamentos, possui também diversos serviços, como mostra a Figura 7.4.

Cada um dos serviços possui só um endereço, só uma URI através da qual pode ser acessado.

É possível alterar essa forma de exposição do sistema ao disponibilizar mais endereços, em geral uma URI para cada entidade (recurso), mas com um número limitado de operações. Cada operação passa a trabalhar com um recurso específico através de sua representação, aumentando a capilaridade. A Figura 7.5 é baseada em um similar de Stefan Tilkov, que tenta descrever tais diferenças[201].
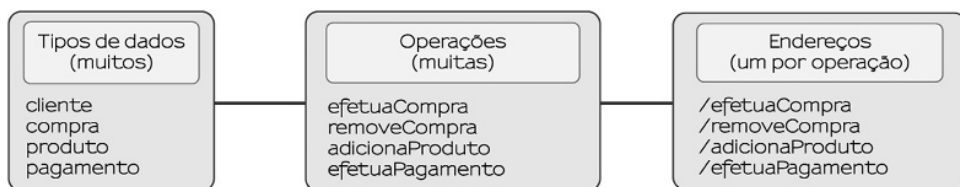
Figura 7.4: Tipos, operações e endereços em uma abordagem orientada a Serviços.
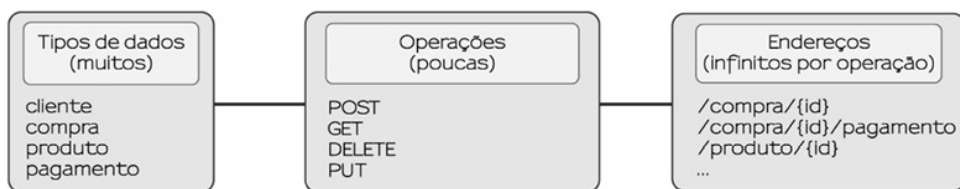
Figura 7.5: Tipos, operações e endereços em uma arquitetura REST.

Cada um desses dados é considerado um recurso (*resource*), algo intangível diretamente, que será acessado e manipulado através de suas representações (*representation*). Por exemplo, um pedido de compra é um recurso, que pode ser encontrado em um endereço (*address*), na URI *http://arquiteturajava.com.br/loja/compras/1535434*, representado através de um XML.

A comunicação está sendo feita entre clientes e servidores através de representações desses recursos, o que forma a base de uma arquitetura orientada a recursos (*Resource Oriented Architecture – ROA*)[199][169]. A adoção desta arquitetura junto a outras características, como uso de uma interface uniforme, stateless, navegação entre seus recursos e estados através de controles como os de hipermídia (*HATEOAS*)[59], caracteriza uma arquitetura REST (*Representational State Transfer*)[211].

Sendo assim, REST pode ser alcançado através de, por exemplo, HTTP e representações com suporte a hipermídia, mas poderia ser utilizada outra pilha de tecnologias com as mesmas características. A vantagem da adoção do HTTP juntamente com XML ou JSON está na onipresença dessas tecnologias. A partir de agora, assumiremos a implementação de uma arquitetura distribuída baseada em hipermídia em cima do protocolo HTTP.

A interface de comunicação (*uniform interface*) do REST também requer a utilização de mensagens autodescritivas, isto é, qualquer intermediário (como um proxy) é capaz de entender a intenção de uma mensagem e tomar uma ação de acordo.

A interação entre um cliente e o servidor através de REST inicia-se através de um ponto de entrada (*entry-point*), uma URI que permite acessar o resto do sistema. Esse novo acesso é realizado através de controles, como simples links:

```
<loja>
    <nome>Loja de livros</name>
    <horarioDeFuncionamento>10:00-22:00</horarioDeFuncionamento>
    <link rel="clientes" href="/clientes" />
    <link rel="produtos" href="/produtos/guia_completo" />
</loja>
```

Eles costumam descrever o relacionamento entre o recurso acessado e outros através do atributo rel, além do endereço presente em href. Resultados de requisições podem ser cacheados pelo cliente, proxies e até mesmo pelo servidor, diminuindo consumo de recursos[3].

Ter uma granularidade pequena como resposta não deve ser motivado por economia de banda, já que uma representação maior, contendo mais informações, também pode ser cacheada. Usando diversos headers[147], é possível diminuir drasticamente o número de requisições e/ou a banda consumida[75].

O *JAX-RS* é a especificação do Java para serviços Web REST, e pode ser utilizada juntamente com outras tecnologias do Java EE, como fazer integração com EJBs. Usando *JAX-RS* é possível descrever as operações de criar e mostrar um plano de saúde, como no código a seguir:

```
@Path(“/planos")
    public class PlanoDeSaudeResource {

        @GET
        @Path(“/{planoId}")
        @Produces(“application/xml")
        public Plano mostrar(@PathParam(“planoId") String planoId) {
            // busca o plano
        }

        @POST
        @Consumes(“application/xml")
        public void criar(JAXBElement<Plano> plano) {
            // armazena o plano
        }
    }
```

Com isso, ao executar um GET em `/planos/15`, a variável `planoId` passa a ter o valor 15, e o método mostrar é responsável por devolver um `Plano` que será serializado em XML de acordo com suas anotações *JAX-B*. Fazendo um POST para /planos e enviando um XML no corpo da requisição, este será desserializado para um `JAXBElement<Plano>`.

Essa primeira geração de ferramentas focadas em REST, incluindo o *Jersey*, aborda somente os aspectos ligados ao protocolo HTTP[203], como mencionado por Leonard Richardson em sua classificação de serviços Web.5 A versão 2 da especificação JAX-RS atende aos quesitos de API cliente e suporte a hipermídia[33]. Frameworks, como *VRaptor*, *Restfulie* e *Restlet*, suportam nativamente a utilização de controles hipermídia como o meio de interação com o servidor.

Clientes e hipermídia

No exemplo de clientes e ordens, o código a seguir utiliza o *Restfulie* para acessar o entry point da loja:

```
Response response = Restfulie.at(“http://arquiteturajava.com.br/loja")
    .accepts(“application/xml").get();
System.out.println(response.getCode());
System.out.println(response.getBody());
```

É possível também que a biblioteca adotada forneça uma API de alto nível para deserialização e interpretação do conteúdo existente dentro de uma representação:

```
String uri = ‘‘http://arquiteturajava.com.br/loja/clientes/15";
Response response = Restfulie.at(uri).accepts(‘‘application/xml").get();
Cliente cliente = response.getResource();
```

Neste exemplo, o cliente conhece um modelo/padrão de URIs (*URI template*)[53], que identifica um conjunto de URIs. As URIs adotadas podem ser informativas para que um ser humano a compreenda[2], mas basear-se em padrões como *http://arquiteturajava.com.br/loja/clientes/$id* tornaria os clientes ainda mais acoplados ao serviço.

Sendo assim, o cliente deve conhecer somente a entrada do sistema,o seu *entry-point*. A partir deste ponto, o servidor dirá ao cliente quais outros recursos poderá acessar através de, por exemplo, um link. Um link possui um valor semântico, um significado, que descreve o relacionamento entre o recurso atual e aquele que será atingido ao acessá-lo. Links são exemplos de controles hipermídia[59], uma marcação que conecta dois recursos dentro de uma rede de maneira não linear[144].

Nas abordagens tradicionais de serviços Web, requisições retornam XML ou JSON, cujo conteúdo é composto somente pelos dados acessados. Seguindo os conceitos de SOA, um primeiro serviço é criado para a busca de clientes, um segundo para permitir efetuar pagamentos, assim como um terceiro para retornar a lista de serviços contratados e ativos de determinado cliente.

Na implementação tradicional de SOAP, seria necessário que o cliente conhecesse de antemão (*early-binding*) todas as URIs que acessará, escrevendo as mesmas dentro do código, decisões em *design time*. Tomando como exemplo a pesquisa de um cliente, o XML a seguir representa um possível retorno:

```
<cliente>
    <nome>José Beltrano</nome>
    <empresa>Loja Caelum</empresa>
</cliente>
```

Neste caso, não há como saber qual URI acessar, nem quais parâmetros enviar, para obter os pagamentos efetuados e serviços contratados por esse cliente. O problema fica ainda mais complicado ao tentar descobrir como executar tarefas de escrita, como efetuar um novo pagamento.

Na Web humana, é diferente. Cada requisição não retorna apenas os dados (o HTML em si) e a estrutura utilizada (o esquema de um HTML), mas também elementos como links e formulários que permitem ao cliente navegar pelo sistema, passando de uma página a outra para atingir seu objetivo. Essas formas de interação com

o servidor podem mudar a cada requisição – novos links, URIs que mudam, etc. Essas páginas não possuem apenas uma representação do recurso, mas também dados para controlar o fluxo do cliente na aplicação (dados de controle, *control data*).

A representação de um cliente que utiliza links como controle hipermídia pode ser feita em diversos formatos de representação, como em HTML:

```
<html>
<body>
<div class="cliente">
<span class="nome">Guilherme Silveira</span>,
    <span class="empresa">Loja Caelum</span>
    </div>
    <div class="navegacao">
    <a href="/cliente/7453687/pagamentos">pagamentos</a> |
    <a href="/cliente/7453687/servicos">serviços</a> |
    </div>
    </body>
    </html>
```

Um cliente genérico, como um navegador, é capaz de apresentar o resultado para o cliente, que tomará a decisão de qual passo seguir através dos controles presentes. Quando a empresa decidir que é melhor controlar os pagamentos através de um outro fornecedor, será suficiente alterar a URI (*href*) para a qual os links apontam. Uma vez que o tipo de relacionamento entre os dois recursos é o mesmo, os links ainda representam os pagamentos e os serviços desse cliente, que continuam funcionando. Em casos extremos, até mesmo o formato da mensagem trocada poderia mudar, um problema resolvido através de *media types*.

A integração de sistemas via hipermídia utiliza esses controles para beneficiar projetos que também fazem a comunicação máquina-a-máquina[54]. Substituindo o HTML por XML, o resultado da busca fica como no exemplo a seguir:

```
<cliente>
    <nome>Guilherme Silveira</nome>
    <empresa>Loja Caelum</empresa>
    <link rel="pagamentos" href="/cliente/745368/pagamentos" />
    <link rel="pagamentos-form" href="/cliente/745368/pagamentos-form" />
    <link rel="servicos" href="/cliente/745368/servicos" />
</cliente>
```

Para gerar um recurso como este, é possível utilizar uma template engine qual-

quer ou maneiras programáticas de configurar os controles hipermídia, como a seguir, no caso do Restfulie:

```
@Get
@Path("/cliente/{cliente.id}")
public void mostra(Cliente cliente) {

    ConfigurableHypermediaResource recurso;
    recurso = restfulie.enhance(database.busca(cliente));
    recurso.relation("pagamentos")
        .uses(PagamentosController.class).lista(cliente);
    recurso.relation("pagamentos-form")
        .uses(PagamentosController.class).form(cliente);
    recurso.relation("servicos")
        .uses(ServicosController.class).lista(cliente);

    result.use(representation()).from(resource, "cliente").serialize();
}
```

Uma segunda requisição GET permite então recuperar os dados dos pagamentos de um cliente, que também é uma representação desses recursos. Note como o link é acessado e navegado; o cliente está acoplado à presença e ao seu significado:

```
Cliente cliente = response.getResource();
response = resource(cliente).getLink("pagamentos").follow().get();
Pagamentos todos = response.getResource();
```

Na Figura 7.6, é possível visualizar como funciona o acesso a uma representação através de um link hipermídia, que relaciona dois recursos distintos na Web.
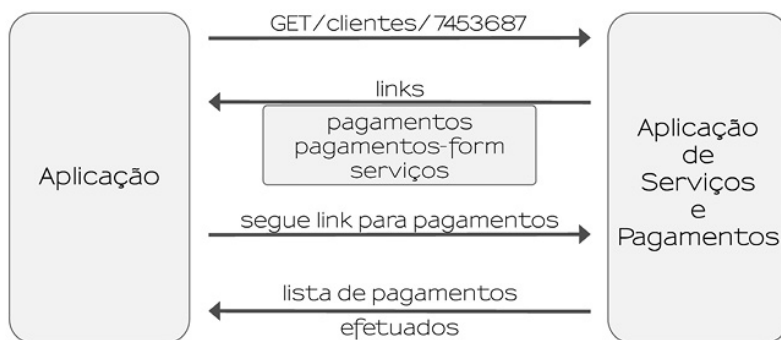
Figura 7.6: O cliente seguindo o link que possui o significado de seus pagamentos.

Caso o servidor decida alterar a máquina, ou até mesmo o fornecedor que controla a listagem de pagamentos, basta que ele mude o link. O servidor passará a indicar ao cliente onde encontrar as informações, em vez de ir buscá-la. O controle é invertido. Como o cliente está acoplado ao significado de um link, e não com a URI, ele se adapta automaticamente, conforme o exemplo ilustrado na Figura 7.7.

Figura 7.7: Devido ao acoplamento semântico com o link, e não com a URI, o cliente acessa o serviço de controle de pagamentos que o servidor utilizar.

É recomendado o uso de namespaces nos atributos `rel`, como em *http://loja.arquiteturajava.com.br/rel/servicos*, para desambiguar possíveis situações em que uma mesma palavra possua dois significados. Repositórios como o IANA permitem o registro do significado de relacionamentos, como é o caso da palavra `payment`.

Enquanto links são uma forma de controle hipermídia, que permite fácil navegação, os formulários proveem uma maneira de interagir com o estado da aplicação para que a aplicação do cliente descubra em tempo de execução quais dados e de que maneira eles devem ser enviados[170]. Uma possível implementação de formulário pode ser feita através da utilização de uma URI específica para obter um exemplo de representação a ser utilizado na próxima requisição, como no código a seguir:

```
<link rel=’’pagamentos-form’’ href=’’/cliente/7453687/pagamentos-form’’ />
```

Formulário remete à identificação de um modelo (*template*), que apresenta quais dados e de que maneira eles devem ser enviados para o servidor. No caso de HTML, por exemplo, a composição das tags a seguir indica ao cliente que algo com a semântica `donoDoCartao` deve ser enviado, com uma ajuda de descrição para um ser humano compreender como "*o nome do dono do cartão de crédito*", além de um valor de exemplo "Guilherme Silveira":

```
<label for=''donoDoCartao''>O nome do dono do cartão</label>
<input type=''text'' name=''donoDoCartao'' id=''donoDoCartao'' value=''Guilherme Silveira'' />
```

Uma aplicação pode usar uma forma de especificação dessas mensagens para indicar ao cliente como um pagamento deve ser enviado. O exemplo a seguir utiliza o *XForms*, uma especificação do W3C. Note como a palavra formulário não aparece, e o conceito de valores (ou campos) a serem enviados é alcançado através de exemplos (instance):

```
<model xmlns=''http://www.w3.org/2002/xforms''>
    <instance>
<!-- exemplo de pagamento para o cliente, um formulário -->
<pagamento>
    <nomeNoCartao>
        Preencha aqui com o nome do dono do cartão
    </nomeNoCartao>
    <numeroDoCartao>4444555566667777</numeroDoCartao>
    <codigo>262</codigo>
    <dataDeExpiracao>10/2080</dataDeExpiracao>
</pagamento>
</instance>
<submission
    resource=''/cliente/7453687/pagamentos''
    method=''post''
    mediatype=''application/xml'' />
</model>
```

Elementos que permitem a navegação (links) ou fornecem exemplos de dados a serem enviados (formulários) podem ser incluídos ou referenciados em qualquer representação, seja ela através de HTML, XML ou JSON[8][217]. Links podem ser até mesmo inclusos através de cabeçalhos HTTP[149], como em casos em que a representação é binária, como uma imagem.

O *OpenSearch*, por exemplo, criado para implementar buscas na Web, utiliza modelos de URI, por meio de um controle hipermídia, que ensinam ao cliente qual URI deve ser acessada para uma busca[52], como por exemplo o padrão *http://arquiteturajava.com.br/loja/produtos/busca?q=\protect\T1\textbraceleftsearchTerms\protect\T1\textbraceright&p=\protect\T1\textbraceleftstartPage\protect\T1\textbraceright*.

Em vez de centralizar todo o controle em um único ponto de execução, usando hipermídia é mais fácil distribuir o comportamento entre várias partes. Desta ma-

neira, realizar um processo passa a envolver diversos agentes que coordenam sozinhos seu trabalho. Adotando SOAP, isso pode ser alcançado com a especificação do *WS-Choreography* e uma linguagem comum, que ajude a definir como os componentes de um serviço coreografado devem se coordenar.

Operações

Ao utilizar o HTTP como uma API, e não apenas para *tunneling*, o desenvolvedor limita a integração a poucas operações que podem ser realizadas em cada recurso.

Por exemplo, ao trabalhar com produtos, é comum mapear cada método HTTP com um tipo de operação diferente. POST e PUT podem ser usados para criação; PATCH[50] ou PUT para alterar; PUT para substituir; GET para leitura; DELETE para remoção; HEAD para obter os cabeçalhos (metadados) de um recurso; e OPTIONS para entender quais operações são aceitas naquele instante para determinado recurso.

Alguns desses métodos pedem o envio de um corpo, como no caso do PUT e POST para criação de um novo recurso, para onde este deve ser enviado por completo. No caso de PATCH, o corpo define um formato de alterações a serem aplicadas em um recurso (um *diff*). Apesar de ser especificado através de uma *RFC*, o método PATCH é o mais recente e ainda não suportado por algumas APIs.

Media types

Considerando a representação abaixo, agora é possível trabalhar com os links de um recurso:

```
<cliente>
    <nome>Guilherme Silveira</nome>
    <empresa>Loja Caelum</empresa>
    <link rel=''pagamentos'' href=''/cliente/745368/pagamentos'' />
    <link rel=''pagamentos-form'' href=''/cliente/745368/pagamentos-form''/>
    <link rel=''servicos'' href=''/cliente/745368/servicos''/>
</cliente>
```

Mas como interpretar este conteúdo ou, até mesmo, saber se através de uma tag link é possível executar uma operação? Para isto, é necessário indicar o significado dele, dar-lhe algum valor semântico. O significado de uma representação trocada entre clientes e servidores é compreendida pelo seu *media type*. Seu objetivo é explicar

como a estrutura e o significado dos dados devem ser interpretados e compreendidos. Durante uma requisição, o cliente e o servidor negociam para escolher um *media type*[102] que ambos compreendam (*content negotiation*) (Figura 7.8).
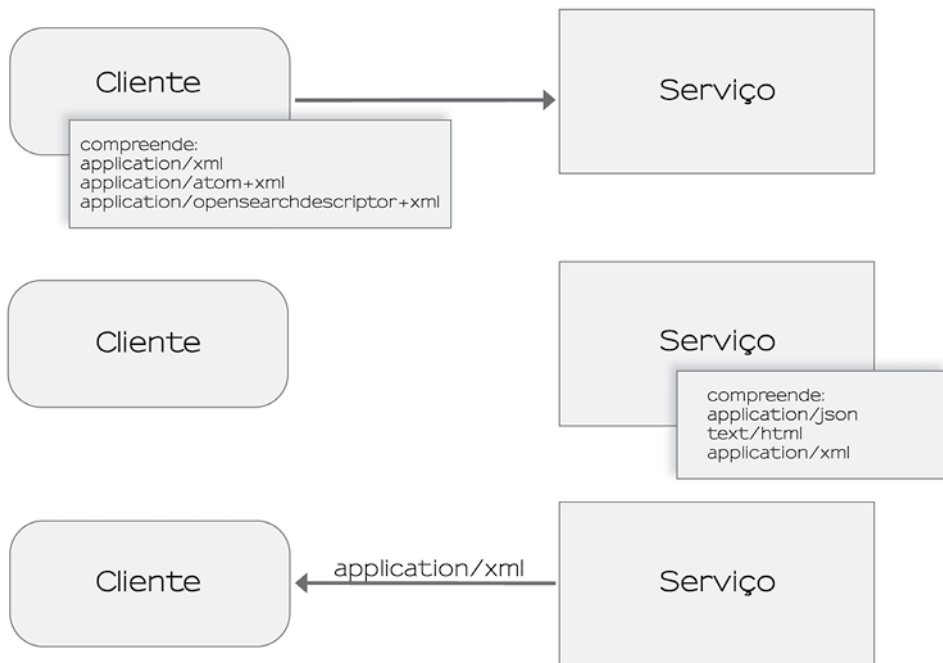
Figura 7.8: Negociação ocorrendo entre cliente e servidor.

Sem conhecer qual o media type utilizado na representação, é impossível afirmar com exatidão se ele é um XML (`application/xml`), uma variação de XML (`application/atom+xml`), ou algum outro formato. Além disso, pela sua definição, um XML não possui links, somente tags; portanto, onde um ser humano interpreta o significado da tag link como um controle hipermídia, uma máquina deveria interpretar como uma outra tag qualquer.

Para extrair o formato e a semântica da mensagem, é necessário que toda representação venha acompanhada do nome do *media type* que foi utilizado. O processo de interpretação de tags só pode ser feito após analisar o cabeçalho HTTP, que indica qual o *media type* utilizado, por exemplo, `text/html`.

Como no navegador, as aplicações REST entendem a sintaxe e a semântica das tags encontradas em uma representação de acordo com seu *media type*, por exemplo,

`text/html`.

Utilizando o HTTP, a escolha de qual media type usar é feita através do pedido do cliente, presente no cabeçalho Accept. É a chamada negociação. O exemplo a seguir mostra uma requisição que aceita tanto `application/xml` quanto `application/json`:

```
GET /cliente/7453687 HTTP/1.1
Host: arquiteturajava.com.br
Accept: application/xml, application/json
```

Na resposta, o cabeçalho `Content-type` indica o *media type* escolhido:

```
HTTP/1.1 200 OK
Date: Sun, 18 Sep 2011 18:09:00 GMT
Content-type: application/xml

<cliente>
<nome>Guilherme Silveira</nome>
<empresa>Loja Caelum</empresa>
<link rel=''pagamentos'' href=''/cliente/745368/pagamentos'' />
<link rel=''pagamentos-form'' href=''/cliente/745368/pagamentos-form''/>
<link rel=''servicos'' href=''/cliente/745368/servicos''/>
</cliente>
```

Nem sempre a resposta ao processo de *content negotiation* é um sucesso. O servidor pode ser incapaz de fornecer uma representação através dos *media-types* aceitos pelo cliente. O HTTP permite a utilização de seus *response codes* para descrever diversas situações importantes na comunicação entre dois sistemas. Por exemplo, quando o servidor não encontra um *media-type* aceitável, ele deve devolver o código `406 Not Acceptable` junto a uma resposta que contenha referências para aqueles que são aceitos.

Na integração de sistemas, os formatos mais comuns são o XML e o JSON. Uma vez que URIs dentro do corpo devem ser tratadas como texto, ambos não definem semântica de controles hipermídia. Para isto, temos outras opções, como a criação de um *media-type* próprio. Por exemplo, uma loja de livros cria `application/vnd.livro+xml`, enquanto uma loja de filmes `vnd/filme+xml`. Uma loja inteira pode criar seu próprio *media-type* como `vnd/minha_livraria+xml`.

O sufixo `*/*+xml` (como `application/vnd.caelum+xml`) é previsto pela RFC 3023[142], permitindo que o desenvolvedor defina as regras de compreensão do con-

teúdo através do conteúdo base, mas ainda, assim, indicando que usa a sintaxe de XML.

Mas cada uma dessas extensões vive isoladamente, ficando impossível uma representação única que inclua informações encontradas em dois *media-types* distintos. Por exemplo, se livros são definidos através de um *media-type*, filmes através de outro, pessoas e recomendações através de um terceiro, fica impossível a criação de uma única representação com todas essas informações. Por isso, são recomendadas abordagens, que permitam qualquer combinação de conteúdo e regras de processamento deles.

Em vez de criar vários *media-types* baseados em XML[130], é possível adotar um único *media-type* mais genérico[23]. As regras de processamento são definidas nesse media type, permitindo a utilização de pequenas definições de semântica e processamento referenciadas dentro da representação, como no caso dos microformatos (*micro-formats*)[218], como *hCalendar* e *hCard*. Eles permitem a definição de um contrato fixo, com garantias para validação e compatibilidade, além de se manter parcialmente flexível, com liberdade para evolução, diminuindo o acoplamento em relação a um *schema* fixo.

Microformatos também auxiliam o SEO de alguns mecanismos de buscas, como pelo Google[44][131] e Yahoo Search Monkey. Mike Amundsen sugere a adoção de XHTML como um *media-type* que suporta controles hipermídia[9]. O código a seguir é um exemplo de local descrito utilizando um microformato através do *media-type* `application/xhtml+xml`:

```
<div class=''location vcard''>
    <span class=''fn org''>Caelum</span>
    <div class=''adr''>
        <div class=''street-address''>Rua Vergueiro, 3185, 8o andar</div>
        <div>
            <span class=''locality''>São Paulo</span>
            <span class=''postal-code''>04101300</span>
        </div>
    </div>
</div>
```

Tecnologias como RDF (*Resource Description Framework*) também visam desacoplar o consumidor de seus serviços[117].

Com o acoplamento do cliente ao *media-type*[211], e não mais a um servidor determinado, **elimina-se a necessidade de especificações como WSDL e WADL, per-**

mitindo a tomada de decisões em tempo de execução, em vez de design time.[87] [10]

Representando conceitos: ontologias

É comum encontrar sistemas que apresentam entidades similares, como pessoas, amigos, músicas, clientes, produtos, seguros, filmes, etc. Por exemplo, o significado de um cartão de crédito é igual em grande parte de aplicações. Mas, se a maneira como tais sistemas apresentam ou requerem tais informações diferir entre si, clientes que desejam utilizá-los terão um maior custo de desenvolvimento e manutenção.

Sem utilizar um padrão amplamente aceito ou conhecido e funcionando somente com os detalhes de determinado serviço, os clientes ficam reféns das decisões tomadas pelos servidores. Um bom exemplo de acoplamento à estrutura está nas APIs de redes sociais como o Twitter, LinkedIn e Facebook. Todos possuem estruturas simples para indicar, por exemplo, relacionamentos entre usuários, mas uma vez que cada um utiliza formatos distintos de XML ou JSON, os clientes são obrigados a escrever um código diferente para acessar e interagir com cada um dos serviços. O código a seguir mostra um exemplo de relacionamento que usa a API do Twitter:

```
{
    ‘‘previous_cursor’’: 0,
    ‘‘ids’’: [
        143206500,
        143201760,
        777920
    ],
    ‘‘previous_cursor_str’’: ‘‘0’’,
    ‘‘next_cursor’’: 0,
    ‘‘next_cursor_str’’: ‘‘0’’
}
```

E o JSON a seguir é a forma escolhida para representar recursos que possuem o mesmo significado, mas desta vez através do Facebook:

```
{
    ‘‘data’’: [
        {
            ‘‘name’’: ‘‘Paulo Silveira’’,
            ‘‘id’’: ‘‘6012890’’
        },
```

```
        {
            ‘‘name’’: ‘‘Sérgio Lopes’’,
            ‘‘id’’: ‘‘6314230’’
        }
    ]
}
```

Apesar do mesmo significado (neste caso, a semântica de relacionamento entre usuários), a não utilização de um conjunto de conceitos padronizados, ou de uma ontologia (*ontology*) para relacionamentos, resulta em clientes frágeis e com custo maior para criação.

Muitos acabam optando pela criação de APIs open source, como clientes genéricos que tratam cada serviço através de um parser distinto, facilitando o trabalho dos desenvolvedores finais. Tais bibliotecas seriam desnecessárias se a representação utilizasse um formato bem conhecido (*well-known*), capaz de dar significado a diversos tipos de relacionamentos, generalizando as duas anteriores, como o exemplo `application/json` a seguir:

```
{
    ‘‘person’’ : {
        ‘‘name’’ : ‘‘guilhermecaelum’’,
        ‘‘relations’’ : [
            ‘‘follows’’ : ‘‘paulo_caelum’’,
            ‘‘follows’’ : ‘‘sergio_caelum’’
        ]
    }
}
```

Utilizar padrões próprios facilita o desenvolvimento a curto prazo, mas implica custos de manutenção mais altos devido ao acoplamento do cliente com um único servidor. Já existem diversos padrões no mercado que podem ser utilizados, como o caso do *friend of a friend* (FOAF)[140], que descreve pessoas e seus relacionamentos.

Code on demand

Outras técnicas permitem a adaptação do cliente de acordo com as novas funcionalidades que um serviço disponibiliza, entregando código sob demanda (*code on demand*). Desta forma, os clientes podem aprender novas maneiras de interagir com o serviço. A utilização de código JavaScript nos navegadores, XMLs de configuração de UI, código compilado como Flash ou Applets são exemplos de *code on demand*.

Diversas questões de segurança surgem ao adotar este tipo de abordagem, uma vez que código será executado no cliente. A Figura 7.9 ilustra uma aplicação para celular que pode construir sua interface com o usuário através de uma representação:
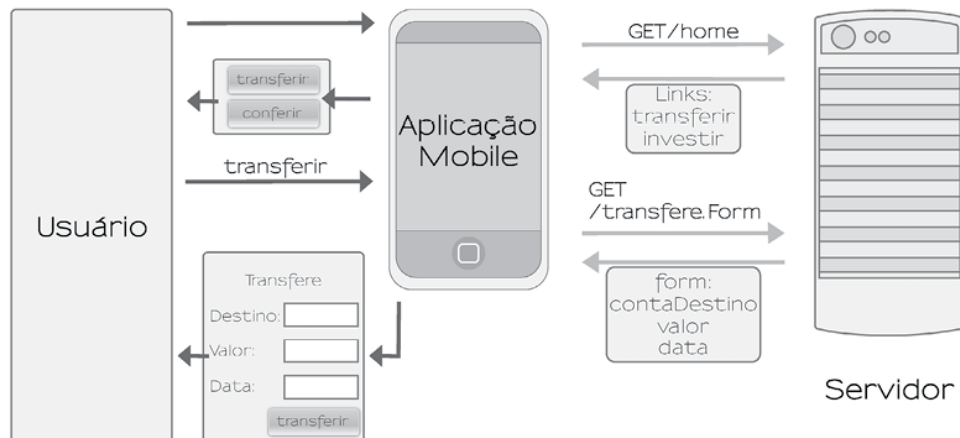
Figura 7.9: Usuário acessando celular com interação definida de acordo com as respostas do servidor.

A utilização de uma representação com código JavaScript, acessando o modelo da aplicação no cliente, permite a customização do comportamento, como no caso comum de *code on demand* encontrado na Web, o JSONP. No exemplo a seguir, a mesma aplicação para celular permite agora mostrar a conta de destino antes de executar a transferência:

```
<exemplo>
    <contaDestino></contaDestino>
    <valor>0</valor>
    <data>13/03/2011</data>
    <script>
        <![CDATA[
            contaDestino.onChange(function() {
            buscaNomeDoDestinatario(contaDestino.val());
            });
        ]]>
    </script>
</exemplo>
```

Apesar de o conceito de hipermídia e de *code on demand* estarem na base de uma arquitetura REST, existem dezenas de sistemas utilizando HTTP para *tunneling*, que obtiveram grande sucesso, seja com XML ou JSON, e não possuem a necessidade de adotar REST.

Seja seguindo um contrato rígido, adotando uma validação parcial dos *schemas*, acessando partes do sistema dinamicamente através de hipermídia ou aprendendo com código sob demanda, o desenvolvedor possui um leque de tecnologias à sua disposição que resolvem o mesmo problema de integração, mas que, a longo prazo, influenciarão a manutenção de seus clientes.

Estado conversacional

Para implementar um carrinho de compras, costuma-se utilizar cookies, e, com isso, a mesma URI (como *http://arquiteturajava.com.br/loja/carrinho*) identifica dois recursos diferentes, dependendo do cliente que o acessa. As mensagens trocadas entre cliente e servidor não são mais autodescritivas (*self describing messages*). A Figura 7.10 exemplifica tal situação.
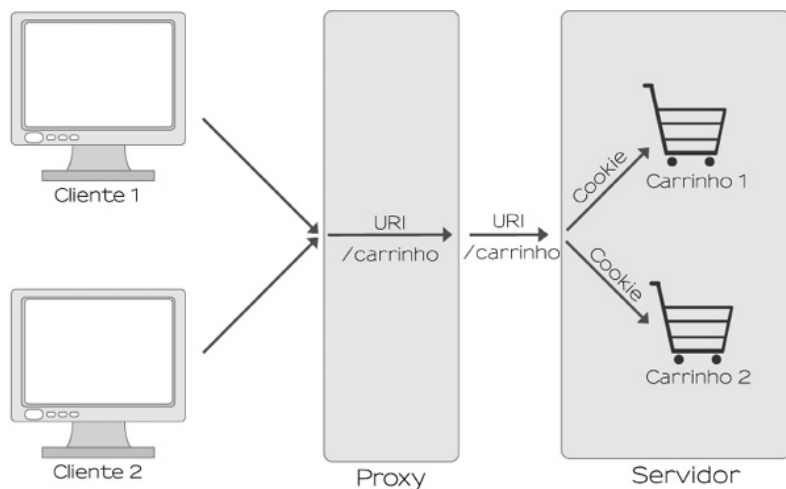
Figura 7.10: Servindo dois recursos diferentes através de uma única URI.

Quando o servidor trata duas requisições para a mesma URI de maneira diferente, dependendo de informações armazenadas no cliente, intermediários perdem a capacidade de entender o que a requisição significa. Este estado conversacional

deve ser evitado para maximizar o potencial de caches e também possibilitar o acesso a mais de um carrinho de compras.

Toda informação que descreve a mensagem e que não faz parte da interface uniforme ou do *media-type*, é chamada de conhecimento fora de banda (*out of band knowledge*)[59]. REST é *stateless* no sentido de não manter estado da aplicação no cliente. Uma das maneiras de transformar uma abordagem que utiliza cookies para uma REST é identificar o carrinho sem quebrar a interface uniforme, como, por exemplo, na URI em *http://arquiteturajava.com.br/loja/carrinho/57389386257*. Ou seja, cada recurso passa a ter sua própria URI.

É parecido com o que os servidores Web permitem fazer com o valor de JSESSIONID na URI, mas utilizando session e cookies apenas para guardar informações que não precisam ser endereçáveis nem afetam o resultado de uma requisição. Isso impossibilita o sequestro de sessão quando há compartilhamento de URIs entre usuários (que poderia acontecer no uso de JSESSIONID), já que os dados relativos a um único usuário podem continuar sendo transmitidos por sessão e cookies.

Uma dificuldade encontrada em Web Sites é a de não poder comprar duas passagens aéreas ou pagar dois boletos bancários em abas diferentes de um mesmo browser, dado o péssimo uso de cookies para expressar o estado atual de um cliente. Utilizando uma abordagem mais amigável ao endereçamento, ganha-se também a possibilidade de ter dois carrinhos de compra para um mesmo usuário ou, ainda, compartilhar suas compras e pagamentos com outros usuários, caso seja interessante para a aplicação.

O exemplo da Figura 7.11 mostra os carrinhos agora com URIs distintas, onde o cache pode ser feito sem problemas e o usuário pode realizar duas compras simultaneamente, ou, ainda, compartilhar suas compras e pagamentos com outros usuários.
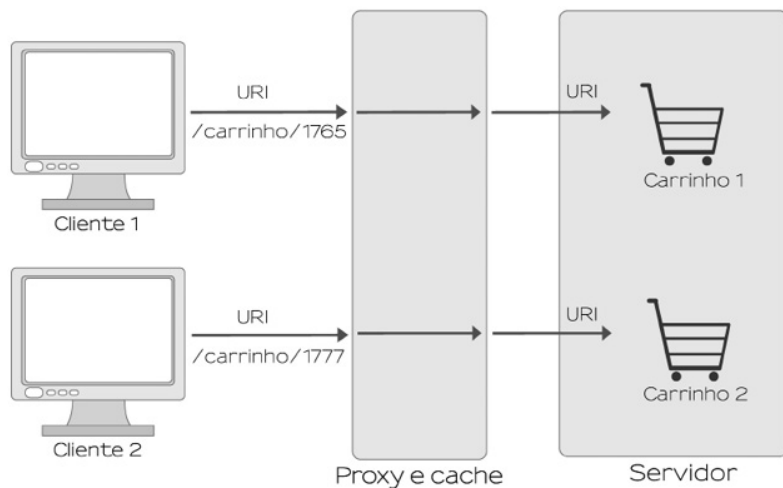
Figura 7.11: Recursos podem ser compartilhados ou cacheados mais facilmente quando cada um possui sua própria URI.

Utilizando cookies e sessão para guardar o identificador do carrinho, seria difícil cachear ou compartilhar o recurso. Problemas podem aparecer até mesmo com o botão de voltar, deixando o servidor confuso com mais de uma aba.

Quando a aplicação não armazena estado do cliente em sessões, é possível adotar um balanceador de carga (*load-balancer*), que não precisa jogar o cliente sempre para o mesmo nó servidor (*sticky session*), nem é necessário transferir sessões de um lado para outro (*session replication*), nem ficar dependente de um único servidor. Isso minimiza o processamento e reduz a quantidade de mensagens trocadas.

Como usar caches em um serviço Web

Caches são intermediários importantes ao utilizar a Web como infraestrutura e plataforma. Eles diminuem a quantidade de dados transferidos e o tempo esperado de uma resposta. No HTTP 1.1, a maior parte das configurações de cache é feita através do header `Cache-Control`, como no exemplo a seguir, que permite 10 minutos de cache em clientes ou intermediários:

```
HTTP/1.1 200 OK
Date: Mon, 21 Feb 2011 10:10:00 GMT
Last-Modified: Sun, 20 Feb 2011 10:10:00 GMT
Cache-Control: max-age=600,public
```

No HTTP, as requisições condicionais, que usam cabeçalhos como `If-Not-Modified`, permitem a um proxy ou servidor somente entregar uma nova resposta caso o recurso tenha sido alterado. Em diversas situações, esses caches ajudam a diminuir o número de requisições (*round-trips*) ou a banda consumida. O exemplo a seguir mostra a requisição de um cliente que possui em seu cache os dados de um recurso, mas deseja verificar se ele não foi alterado desde uma data específica:

```
GET /clientes/743756 HTTP/1.1
Host: arquiteturajava.com.br
If-Modified-Since: Sun, 20 Feb 2011 10:10:00 GMT
```

A resposta do servidor, caso não tenha sido alterado, evita o consumo desnecessário de banda:

```
HTTP/1.1 304 Not Modified
Date: Mon, 21 Feb 2011 10:10:00 GMT
Last-Modified: Sun, 20 Feb 2011 10:10:00 GMT
Cache-Control: max-age=3600,must-revalida
```

Uma prática comum consiste em colocar uma aplicação (*proxy reverso*) funcionando como cache na máquina do servidor. Ele pode armazenar todas as representações públicas e cacheáveis, evitando o processamento caso elas sejam acessadas por diversos clientes. É possível beneficiar-se de caches de maneira segura em HTTP, mesmo quando o cliente precisa estar autorizado. Mais importante, os caches de requisições HTTP estão onipresentes. Diferente de caches específicos ao server side, eles podem aparecer em navegadores, APIs clientes, servidores, proxies, etc. (Figura 7.12).
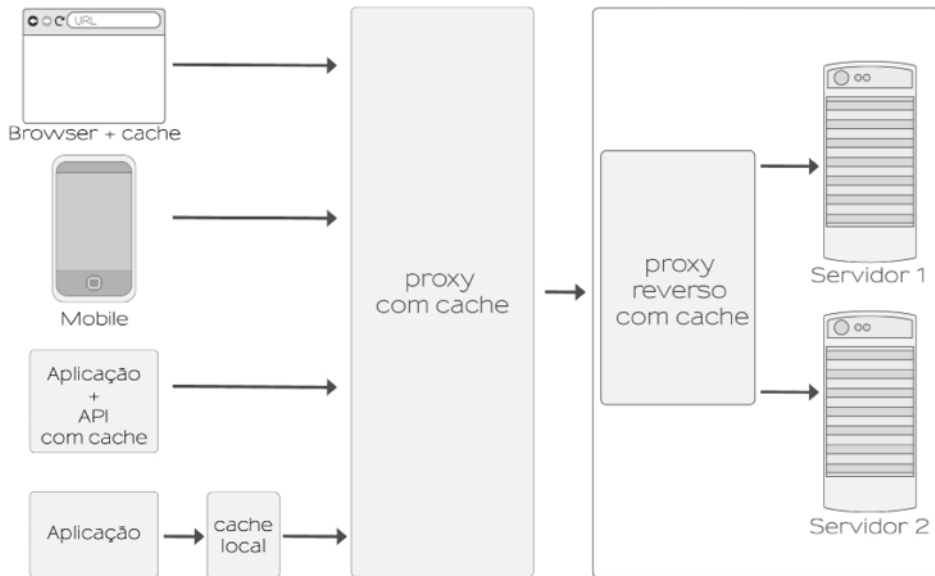
Figura 7.12: Caches HTTP podem aparecer tanto em clientes quanto em servidores e intermediários.

A resposta de um servidor não varia somente de acordo com a URI, mas também com os cabeçalhos da requisição HTTP. Para não quebrar caches, o servidor deve indicar quais cabeçalhos da requisição influenciaram em seu resultado. Para uma requisição que utilizou o cabeçalho `Accept: application/xml`, o resultado a seguir faria com que uma proxy entregasse o mesmo XML para uma requisição com `Accept: application/json`:

```
GET /clientes/743756 HTTP/1.1
Host: arquiteturajava.com.br
Accept: application/xml

HTTP/1.1 200 Ok
Date: Mon, 21 Feb 2011 10:10:00 GMT
Last-Modified: Sun, 20 Feb 2011 10:10:00 GMT
Cache-Control: max-age=3600,must-revalidate
Content-Type: application/xml
```

O mesmo ocorre com outros headers de negociação, como o `Accept-Encoding`, para suportar compressão gzip. O servidor deve utilizar o cabeçalho `Vary` para re-

solver o problema, como `Vary: Accept`.

Da mesma maneira que, na Web humana, é possível quebrar uma única requisição em diversas para otimizar o uso de cache[75], sistemas forçarão múltiplas requisições na API para recuperar pequenos pedaços de informação. Essas práticas de quebrar requisições grandes em menores e utilizar hipermídia para navegar no sistema permitem otimizações de cache e podem ser adotadas por qualquer sistema integrado através da Web. Repare que é uma abordagem diferente da que costuma ser aplicada com SOAP, onde uma granularidade maior é recomendada, evitando um número grande de requisições.

Tomando como exemplo um serviço que retorna um relatório com informações financeiras detalhadas, é possível quebrar a requisição, para que elas sejam feitas baseadas no parâmetro do ano. Se o cliente necessitar da informação de 4 anos diferentes, fará 4 requisições. A vantagem será que cada uma dessas requisições poderá ser cacheada, e até mesmo ter tempos de expiração diferentes. Aproveita-se do fato de que o relatório financeiro dos anos anteriores provavelmente não mudarão, e ver dados stale não é um grande problema. Se fosse apenas uma única requisição, não seria possível se aproveitar tanto assim do cache.

Uma solução mais elaborada pode ter como entry point o relatório do ano corrente, que será entregue junto com um link para os resultados do ano anterior. Os headers indicam um possível período curto de cache. Ao seguir o link, os headers indicam um período de cache muito maior e links para navegação (Figura 7.13):
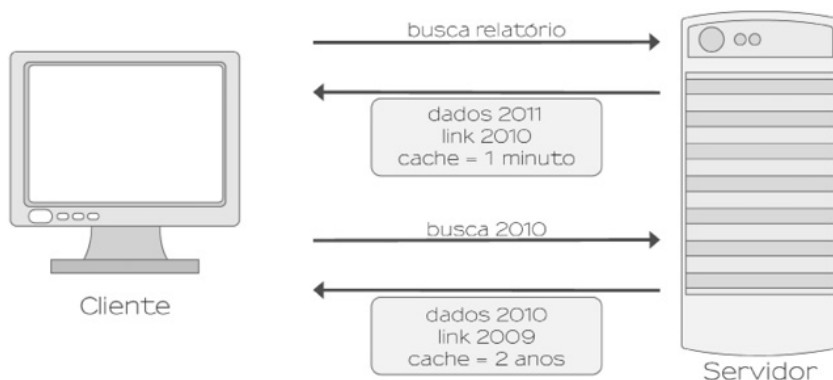
Figura 7.13: Relatório beneficiando-se de cache devido ao uso de hipermídia na quebra de um recurso por tempo.

```
HTTP/1.1 200 OK
```

```
Date: Sun, 18 Sep 2011 18:09:00 GMT
Cache-control: max-age: 63072000
Content-type: application/xml

<pagamentos>
<pagamento>...</pagamento>
<pagamento>...</pagamento>
<pagamento>...</pagamento>
<link rel=''next''
href=''http://arquiteturajava.com.br/loja/pagamentos/2011'' />
<link rel=''previous''
href=''http://arquiteturajava.com.br/loja/pagamentos/2009'' />
</pagamentos>
```

Segmentar o recurso por tempo é uma prática simples e efetiva de *segmentation by freshness*[75], e usar hipermídia é um passo adiante que permite a otimização do cache.

Em algumas situações, o cache pode ser usado como uma forma de prolongar a disponibilidade de serviços que temporariamente estão fora do ar. Utilizando o header de cache-control e seu stale-if-error [148], é possível indicar que uma resposta pode ser entregue do cache, mesmo que este esteja expirado. E, se o servidor não estiver disponível, é uma troca consciente entre mostrar um dado possivelmente desatualizado ou quebrar uma aplicação:

```
HTTP/1.1 200 OK
Cache-Control: max-age=6000, stale-if-error=12000
Content-Type: application/json
```

Através do *stale-while-revalidate*, da mesma RFC, é possível minimizar os danos do efeito dog pile em aplicações servidas na Web que utilizam caches HTTP[88]. Combinar cabeçalhos e técnicas de cache em HTTP é fundamental para o bom uso da Web como plataforma.

Referências Bibliográficas

[1] *Defining Software architecture*. 2009.

[2] Subbu ALLAMANJARU. Restful uris. 2010.

[3] Subbu ALLAMANJARU. *RESTful Web Services Cookbook*. 2010.

[4] Deepak ALUR, Dan Malks, and John Crupi. Core j2ee patterns: Best practices and design strategies. 2. ed. 2003.

[5] Magnus ALVESTAD. Memory leaks where the classloader cannot be garbage collected.

[6] Scott Ambler. The object-relational impedance mismatch.

[7] Scott Ambler. Introduction to test driven design (tdd). 2003.

[8] Mike AMUNDSEN. Designing a hypermedia api with json. 2010.

[9] Mike AMUNDSEN. Restful profiling : Xhtml-based web apis. 2010.

[10] Mike AMUNDSEN. Why wadl when you can run? 2010.

[11] Diane JORDAN an John EVDEMON. Web services business process execution language version 2.0. 2007.

[12] Gregory R. ANDREWS. *Foundations of Multithreaded, Parallel, and Distributed Programming*. 2000.

[13] Abel Avram. Practicing agility in application architecture. 2008.

[14] Roland Barcia. Tired of hand coding jdbc? use ibatis as a data mapping framework instead. 2005.

[15] Kent Beck. Test-driven development: by example. 2002.

[16] Joe BLANDY and Joshua BLOCH. Why doesn't java have an immutable date class? 2003.

[17] Joshua BLOCH. *Effective Java. 2 ed.* 2008.

[18] Joshua BLOCH. *Effective Java. 2. ed.* 2008.

[19] JAVA BLUEPRINTS. Model-view-controller. 2000.

[20] Jonas BONÉR. Real-world scala: Dependency injection (di).

[21] Grady BOOCH. *Object-oriented analysis and design with applications.* 1993.

[22] Evan BOTTCH. Continuous integration – single code line. 2010.

[23] Tim Bray. Don't invent xml languages. 2006.

[24] Adrian BROCK. Jboss classloader history.

[25] Don Brown. My history of struts 2. 2006.

[26] Julian BROWNE. Brewer's cap theorem. 2009.

[27] Steve BURBECK. Applications programming in smalltalk-80™: How to use model-view-controller (mvc). 1987.

[28] Phillip CALÇADO. Internal data transfer objects.

[29] Phillip CALÇADO. Objetos fantoches.

[30] Phillip CALÇADO. How to write a repository. 2010.

[31] Phillip Calçado. How to write a repository. 2010.

[32] Nicholas CARR. *The Big Switch: Rewiring the World, from Edison to Google.* 2008.

[33] Roberto CHINNICI and Marek POTOCIAR. Jax-rs 2.0: The java api for restful web services. 2011.

[34] Mandy Chung. Monitoring and managing java se 6 platform application. 2006.

[35] COMPUTERWORLD. A to z of programming languages: Smalltalk-80. 2010.

[36] Microsoft Corporation. Extension methods (c programming guide).

[37] ORACLE CORPORATION. Ergonomics in the 5.0 java™ virtual machine.

[38] ORACLE CORPORATION. Frequently asked questions about the java hotspot vm.

[39] Oracle Corporation. The history of java technology.

[40] ORACLE CORPORATION. Javadoc do java.lang.ref.

[41] ORACLE CORPORATION. Javadoc do método valueof(int) da classe integer.

[42] ORACLE CORPORATION. Javadoc do system.gc().

[43] ORACLE CORPORATION. Tuning the memory management system.

[44] Mike DANYLCHUK and Nitin SHETTI. Introducing a new rich snippets format: Events. 2010.

[45] Joseph DARCY. Project coin: Updated arm spec. 2010.

[46] Linda DEMICHIEL. Java persistence 2.0 proposed final draft. 2009.

[47] Edsger W. DIJKSTRA. On the role of scientific thought. 1974.

[48] JBOSS DOCUMENTATION. Classcastexceptions – i'm not your type.

[49] Jim DRISCOLL. Servlet history. 2005.

[50] L. DUSSEAULT, Linden LAB, and J. SNELL. Patch method for http. 2010.

[51] Ammon H. Eden and Rick Kazman. Architecture, design, implementation. *25th International Conference on Software Engineering – ICSE*, 2003.

[52] DeWitt et al Clinton. Open search specification. 2000.

[53] J. et al GREGORIO. Uri template. 2010.

[54] Savas et al PARASTATIDIS. The role of hypermedia in distributed system development. *WS-REST '10 Proceedings of the First International Workshop on RESTfulDesign*, 2010.

[55] Roy et al. SINGHAM. *The Thoughtworks Anthology: essays on Software Technology and Innovation*. 2008.

[56] Stefan et al. TILKOV. Soa manifesto. 2009.

[57] Eric Evans. *Domain-driven design: Tackling complexity in the heart of software*. 2003.

[58] Michael FEATHERS. The deep synergy between testability and good design. 2007.

[59] Roy FIELDING. Architectural styles and the design of network based software architectures. 2000.

[60] Neal FORD. Evolutionary architecture and emergent design: Evolutionary architecture. 2010.

[61] Martin Fowler. Anemic domain model.

[62] Martin Fowler. Domain-specific language.

[63] Martin Fowler. Errant architectures.

[64] Martin Fowler. First law of distributed object design.

[65] Martin FOWLER. Inversion of control containers and the dependency injection pattern.

[66] Martin FOWLER. *Refactoring: Improving the Design of Existing Code*. 1999.

[67] Martin FOWLER. *Patterns of Enterprise Application Architecture*. 2002.

[68] Martin Fowler. *Patterns of Enterprise Application Architecture*. 2003.

[69] Martin Fowler. *UML Distilled: a brief guide to the standard object modeling language*. 2003.

[70] Martin Fowler. Who needs an architect? *IEEE Software, v. 20*, 2003.

[71] Martin Fowler. Is design dead? 2004.

[72] Martin Fowler. Fluent interface. 2005.

[73] Martin Fowler. Gettereradication. 2006.

[74] Martin FOWLER. Businessreadabledsl. 2008.

[75] Martin FOWLER. Segmentation by freshness. 2008.

[76] Martin FOWLER. Blue green deployment. 2010.

[77] Martin FOWLER. 2003. Anemic domain model.

[78] Steve FREEMAN and Nat PRYCE. *Growing object-oriented software, guided by tests*. 2009.

[79] Erich GAMMA, Richard HELM, Ralph JOHNSON, and John M. VLISSIDES. *Design Patterns: elements of reusable object-oriented software*. 1994.

[80] Erich GAMMA, Richard HELM, Ralph JOHNSON, and John M. VLISSIDES. *Design Patterns: elements of reusable object-oriented software*. 1994.

[81] Simson GNANAM. How inheritance works in hibernate. 2008.

[82] Brian Goetz. Java theory and practice: To mutate or not to mutate? 2003.

[83] Brian GOETZ. *Java Concurrency in Practice*. 2006.

[84] GOOGLE. Protocol buffers.

[85] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java™ Language Specification. 3. ed.* 2005.

[86] Derek GOTTFRID. Self-service, prorated supercomputing fun! 2007.

[87] Joe GREGORIO. Do we need wadl? 2007.

[88] Ilya GRIGORIK. Asynchronous http cache validations. 2008.

[89] Paul HAMMANT. Inversion of control history.

[90] David HAYDEN. Web applications: N-tier vs. n-laye. 2005.

[91] Eric HERMAN. Composition vs. inheritance in java.

[92] JBOSS HIBERNATE. Batch processing.

[93] JBOSS HIBERNATE. Configuração do c3po no hibernate.

[94] JBOSS HIBERNATE. Improving performance.

[95] JBOSS HIBERNATE. Query cache. 2009.

[96] Eelco HILLENIUS. Wicket is not suited for websites. 2011.

[97] Todd HOFF. Strategy: Break up the memcache dog pile. 2009.

[98] Todd HOFF. Netflix: Use less chatty protocols in the cloud - plus 26 fixes. 2010.

[99] Gregor HOHPE and Bobby WOLF. *Enterprise integration patterns: designing, building, and deploying messaging solutions.* 2003.

[100] Allen HOLUB. Why getter and setter methods are evil. 2003.

[101] Jez HUMBLE and David FARLEY. Continuous delivery: reliable software releases through build, test, and deployment automation. 2010.

[102] IANA. http://tools.ietf.org/html/rfc5789. 2007.

[103] IBM. Dpd chronology.

[104] INNOQ. Web services standards as of q1 2007. 2007.

[105] Ws-* is to REST as Theory is to Practice. 2007.

[106] Nick KALLEN. Big data in real-time at twitter. 2010.

[107] Fabio Kung. Ruby and dependency injection in a dynamic world. 2010.

[108] Ramnivas LADDAD. *AspectJ in action: Enterprise AOP with Spring Applications. 2. ed.* 2009.

[109] Craig Larman. Protected variation: The importance of being closed. *IEEE Software, v. 18*, 2001.

[110] Craig LARMAN. Applying uml and patterns: an introduction to object-oriented analysis and design and iterative development. 3. ed. 2004.

[111] Craig Larman. Bas design architecture. *Practices for Scaling Lean and Agile Development*, 2010.

[112] Julien LECOMTE. Yahoo ui compressor and java class loader.

[113] Tim Lindholm. *The Java™ Virtual Machine Specification. 2. ed.* 1999.

[114] Tim LINDHOLM and Frank YELLIN. The java virtual machine specification: Loading, linking, and initializing.

[115] Sérgio Lopes. Vivendo no cloud: a infraestrutura externa da caelum em 11 soluções. 2011.

[116] Subraya MALLYA. Netflix's move to amazon ec2 – explained. 2010.

[117] Frank MANOLA and Eric MILLER. Rdf primer. 2004.

[118] Robert C. Martin. Dependency injection inversion.

[119] Robert C. Martin. Oo design quality metrics - an analysis of dependencies. 1994.

[120] Robert C. Martin. The single responsibility principle. 1995.

[121] Robert C. MARTIN. The dependency inversion principle. 1996.

[122] Robert C. Martin. The interface segregation principle. 1996.

[123] Robert C. MARTIN. The liskov substitution principle. 1996.

[124] Robert C. Martin. The open-closed principle. 1996.

[125] Robert C. Martin. Stability. 1997.

[126] Robert C. Martin. Design principles and design patterns. 2000.

[127] Robert C. MARTIN. *Agile Software Development: Principles, Patterns, and Practices*. 2002.

[128] Robert C. MARTIN. Railsconf 2010 keynote – cucumber scripts and testing business rules through the ui. 2010.

[129] Jon MASAMITSU. Our collectors (sun).

[130] MICROFORMATS. Plain old xml considered harmful. 2008.

[131] MICROFORMATS. Google adds support for hcalendar and hrecipe rich snippets. 2010.

[132] SUN MICROSYSTEMS. Core j2ee patterns front controller.

[133] SUN MICROSYSTEMS. Código fonte da classe string.

[134] SUN MICROSYSTEMS. The garbage-first garbage collector.

[135] SUN MICROSYSTEMS. Java se 6 java hotspot™ virtual machine garbage collection tuning.

[136] SUN MICROSYSTEMS. Jaxp compatibility guide for the j2se 5 platform.

[137] SUN MICROSYSTEMS. Memory management in the java hotspot™ virtual machine. 2006.

[138] SUN MICROSYSTEMS. Java persistence blueprints. 2007.

[139] SUN MICROSYSTEMS. Core j2ee patterns session façade. http://java.sun.com/blueprints/corej2eepatterns/Patterns/SessionFacade.html.

[140] Libby MILLER and Dan BRICKLEY. The friend of a friend project. 2000.

[141] Hans Muller. Official: Swing is the dominant gui toolkit. 2005.

[142] M. Murata, S. ST. LAURENT, and D. Kohn. Xml media types. 2001.

[143] Glenford MYERS. The art of software testing, 2. ed. 2004.

[144] T. H. NELSON. Complex information processing: a file structure for the complex, the changing and the indeterminate. *ACM '65 Proceedings of the 1965 20th national conference*, 1965.

[145] Dan NORTH and Aslak HELLESOY. Picocontainer design patterns.

[146] Dan NORTH and Aslak HELLESOY. Picocontainer design patterns.

[147] Mark NOTTINGHAM. Caching tutorial for web authors and webmasters. 1998-2010.

[148] Mark NOTTINGHAM. Http cache-control extensions for stale content. 2010.

[149] Mark NOTTINGHAM. Web linking. 2010.

[150] David ORCHARD. Versioning xml vocabularies. 2003.

[151] Larry O'BRIEN. Design patterns 15 years later: An interview with erich gamma, richard helm, and ralph johnson. 2009.

[152] Rajiv PANT. Organizing a web technology department.

[153] Vincent PARTINGTON. Jpa implementation patterns: Data access objects. 2009.

[154] Vincent PARTINGTON. Jpa implementation patterns: Service facades and data transfers objects. 2009.

[155] Anthony PATRICIO. A short primer on fetching strategies. 2010.

[156] Srini PENCHIKALA. Jdbc 4.0 enhancements in java se 6. 2006.

[157] PICOCONTAINER. Picocontainer site.

[158] Lain Pranshu. *Layers and Tiers*. 2006.

[159] Bruno R. Preiss. Mark-and-sweep garbage collection.

[160] Tony PRINTEZIS. Garbage collection in the java hotspot™ virtual machine (generational algorithm). 2004.

[161] Tony PRINTEZIS. How to handle java finalization's memory-retention issues. 2007.

[162] Dan PRITCHETT. Base: An acid alternative. 2008.

[163] Nat PRYCE. "dependency injection"considered harmful. 2011.

[164] Mark RICHARDS, Richard MONSON-HAEFEL, and David A. CHAPPELL. *Java Message Service. 2. ed.* 2009.

[165] Mark RICHARDS, Richard MONSON-HAEFEL, and David A. CHAPPELL. *Java Message Service. 2. ed.* 2009.

[166] Mark RICHARDS, Richard MONSON-HAEFEL, and David A. CHAPPELL. *Java Message Service. 2. ed.* 2009.

[167] Mark RICHARDS, Richard MONSON-HAEFEL, and David A. CHAPPELL. *Java Message Service. 2. ed.* 2009.

[168] Leonard RICHARDSON. Justice will take us millions of intricate moves. 2008.

[169] Leonard RICHARDSON and Sam RUBY. Restful web services. 2007.

[170] Ian ROBINSON. Using typed links to forms. 2010.

[171] Douglas C. SCHMIDT and Tim HARRISON. Double-checked locking: An optimization pattern for efficiently initializing and accessing thread-safe objects. 1996.

[172] Alec SHARP. *Smalltalk by example: the developer's guide*. 1997.

[173] James SHORW. *The Art of Agile Development*. 2007.

[174] Randy SHOUP. ebays architectural principles. 2008.

[175] David SIEGEL. *Pull: the power of the Semantic Web to transform your business*. 2009.

[176] Guilherme Silveira. Compondo seu comportamento: herança, chain of responsability e interceptors. 2010.

[177] Guilherme Silveira. Deploy contínuo: pois integração contínua não basta. 2010.

[178] Guilherme SILVEIRA. Branches e integração contínua: o problema de feature branches. 2011.

[179] Guilherme SILVEIRA. Martin fowler and jez humble on continuous delivery. 2011.

[180] Paulo Silveira. Java 6, as apis de xml, webservices e classloaders. 2007.

[181] Paulo Silveira. Jpa com hibernate: herança e mapeamentos. 2007.

[182] Paulo Silveira. Os 7 hábitos dos desenvolvedores hibernate e jpa altamente eficazes. 2008.

[183] Paulo Silveira. 2009. 2009.

[184] Paulo Silveira. Enfrentando a lazyinitializationexception no hibernate. 2009.

[185] Paulo Silveira. Guj: Acesso ao entitymanager: através de um dao ou diretamente? 2009.

[186] Paulo Silveira. Trabalhando com closures no java 8. 2011.

[187] James SNELL. Restful rpc. 2010.

[188] John SOWA. Sowa's law of standards. 1991.

[189] GOOGLE PAGE SPEED. Web performance best practices.

[190] Joel Spolsky. Don't let architecture astronauts scare you. 2001.

[191] Joel Spolsky. Language wars. 2006.

[192] Bernie SUMPTION. Inheritance is evil, and must be destroyed. 2007.

[193] Richard E. Sweet. The mesa programming environment. *Proceedings of the ACM SIGPLAN 85 symposium on Language issues in programming environments*, 1985.

[194] Attila SZEGEDI. A day in the life of a memory leak hunter. 2005.

[195] Andrew S. TANENBAUM. *Computer networks. 2. ed.* 1988.

[196] Gil TENE. Azul's pauseless garbage collector. 2010.

[197] Royans K. THARAKAN. Brewer's cap theorem on distributed systems. 2010.

[198] Eric THOLOMÉ. Well earned retirement for soap search. 2009.

[199] Stefan TILKOV. *REST und HTTP: Einsatz der architektur des web für integrationsszenarien*. 2009.

[200] Stefan TILKOV. Einstieg in rest. 2010.

[201] Stefan TILKOV. Einstieg in rest. 2010.

[202] Stefan TILKOV. Enterprise it vs. www. 2010.

[203] Stefan TILKOV, Marc HADLEY, and Paul SANDOZ. Jsr 311 final: Java api for restful web services. 2008.

[204] Robert Tolksdorf. Programming languages for the java virtual machine jvm and javascript.

[205] Bill VENNERS. Inheritance versus composition: Which one should you choose? 1998.

[206] Bill VENNERS. Design principles from design patterns – a conversation with erich gamma, part iii. 2005.

[207] Steve Vinoski. Welcome to "the functional web". 2009.

[208] Lan VUONG. Extreme transaction processing patterns: Write-behind caching. 2009.

[209] W3C. Simple object access protocol (soap) 1.1. 2000.

[210] W3C. Soap version 1.2 part 0: Primer (second edition). 2007.

[211] Jim WEBBER, Savas PARASTATIDIS, and Ian ROBINSON. *REST in practice: Hypermedia and systems architecture*. 2010.

[212] EHCACHE WEBSITE. Hibernate caching.

[213] GOOGLE APP ENGINE WEBSITE. Choosing a datastore for google app engine.

[214] TOMCAT WEBSITE. Tomcat manager how to. http://tomcat.apache.org/tomcat-7.0-doc/manager-howto.html.

[215] Simon Willison. A re-introduction to javascript. 2006.

[216] Galder ZAMARREÑO. Using infinispan as jpa/hibernate second level cache provider. 2010.

[217] K. ZYP and G. Court. A json media type for describing the structure and meaning of json documents. 2009-2010.

[218] Tantek ÇELIK. Facebook adds hcalendar and hcard microformats to millions of events. 2011.